普通高等教育“十三五”规划教材
新工科建设之路·计算机类专业规划教材

Python3

从入门到实战

● 董洪伟 编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

VISIT…

内 容 简 介

本书是一本语法与实践相结合的 Python 入门教程，全书分为上、下篇。上篇为“Python 语法与实践”，以简明的语言、易懂的案例介绍 Python 的变量与对象、运算符与表达式、控制语句、函数、内置数据类型，Python 的面向对象特征，如类与对象、派生类、类的实例与静态方法等 Python 语言的核心语法，以及迭代器与可迭代对象、闭包、装饰器、@property、深拷贝与浅拷贝等高级语言特征，还介绍了错误与异常、调试。在核心语法部分采用来自数据结构、游戏编程、信息管理、机器学习、强化学习等其他学科和领域的一些经典问题作为实战演练，展示了 Python 解决实际问题的强大功能，以提高初学者的实际编程能力，使其尽快熟悉语法的使用。下篇为“Python 标准库”，对常用的一些 Python 标准库，如操作系统接口模块、时间日期模块，以及正则表达式、并发计算、图形用户接口编程、网络套接字编程、Internet 应用编程、数据持久化等进行了介绍。

本书描述精练、通俗易懂，提供了丰富的实战案例，既可作为大学本科和高职高专相关专业课程的教材，也可供编程爱好者学习和参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Python3 从入门到实战 / 董洪伟编著. —北京：电子工业出版社，2020.1
ISBN 978-7-121-35356-7

I. ①P… II. ①董… III. ①软件工具－程序设计－高等学校－教材 IV. ①TP311.561

中国版本图书馆 CIP 数据核字(2018)第 251247 号

策划编辑：戴晨辰
责任编辑：张　慧
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　印张：26.75　字数：684.80 千字
版　　次：2020 年 1 月第 1 版
印　　次：2020 年 1 月第 1 次印刷
定　　价：75.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010)88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：dcc@phei.com.cn。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

前言 Preface

Python 是一种易于学习、功能强大的编程语言。它具有高效的高级数据结构，能够简单有效地实现面向对象编程。Python 简单的语法和动态类型，连同解释型特性，使其成为不同平台上脚本处理及快速应用开发的理想语言。此外，Python 还是数据分析和人工智能的首选编程语言。

编著者希望编写一本既简明扼要，又深入全面的 Python 教材，既避免过多的语法细节，又注重语言本身实际使用能力的培养。

本书突出重点，讲解主要的常用语法，而不是面面俱到的语法细节。全书由浅入深，由易到难，尽量用浅显易懂的例子说明语法概念，力求简明扼要，避免空洞的概念和冗长的描述，帮助无编程基础的初学者在较短的时间里快速理解 Python 语言的核心特征。

只有通过具体、长期的实战训练，才能逐步精通一种编程语言。语法知识可以在短期内学习并理解，但只有经过大量实战训练才能真正熟练掌握一种编程语言。本书准备了游戏编程、信息管理、数据结构、机器学习、强化学习等不同领域的经典实战案例，希望可以通过这些案例，帮助读者消化语法知识、提高学习兴趣，逐步将 Python 用于解决各种实际问题而不是用于简单的语法练习，希望避免“只会考试而不会编程”的普遍问题。

实战案例涉及一些其他学科的专业知识，初学者或教师可以根据自己的需要选读或选讲实战部分。

本书包含配套学习资源，读者可在本书的 github 网站（https://hwdong-net.github.io）或登录华信教育资源网（www.hxedu.com.cn）注册后免费下载。

由于编著者水平所限，书中错误之处在所难免，欢迎读者对本书进行批评与指正，共同完善本书内容，使更多的读者受益。

编著者

電子工業出版 PUBLISHING HOUSE OF ELECTRONICS I

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

目录 Contents

上篇　Python 语法与实践

第 1 章　Python 介绍 …… 3
1.1　程序与编程语言 …… 3
1.1.1　计算机是什么 …… 3
1.1.2　计算机编程 …… 4
1.1.3　编译器、解释器和 Python 语言 …… 5
1.1.4　Python 程序开发步骤 …… 6
1.2　Python 语言的主要特征 …… 6
1.3　Python 开发环境及安装方式 …… 7
1.3.1　安装 Python …… 7
1.3.2　Python 开发环境 …… 8
1.4　Python 解释器 …… 8
1.4.1　交互式解释执行模式 …… 9
1.4.2　函数 …… 9
1.4.3　运行脚本文件模式 …… 11
1.4.4　语句和注释 …… 11
1.5　数和字符的表示 …… 12
1.5.1　数的表示 …… 12
1.5.2　字符的表示 …… 13
1.6　如何获得帮助 …… 14
1.7　习题 …… 14

第 2 章　Python 基本计算 …… 16
2.1　值、类型、对象 …… 16
2.1.1　值和类型 …… 16
2.1.2　对象 …… 17
2.2　变量 …… 18
2.2.1　什么是变量 …… 18
2.2.2　变量名和关键字 …… 20
2.2.3　动态类型语言 …… 20
2.3　数据类型概述 …… 20
2.3.1　数值类型 …… 21
2.3.2　列表 …… 21
2.3.3　元组 …… 22
2.3.4　字符串 …… 23
2.3.5　集合 …… 25
2.3.6　字典 …… 25
2.4　类型转换和输入 …… 26
2.4.1　隐式类型转换 …… 26
2.4.2　显式类型转换 …… 26
2.4.3　输入 …… 27
2.5　运算符和表达式 …… 28
2.5.1　运算符和表达式的应用方法 …… 28
2.5.2　运算符的种类 …… 29
2.5.3　运算符的优先级 …… 34
2.6　可变对象和不可变对象 …… 35
2.7　控制语句 …… 38
2.7.1　if 条件语句 …… 38
2.7.2　循环语句 …… 40
2.7.3　pass 语句 …… 43
2.8　实战 …… 43
2.8.1　二分查找 …… 43
2.8.2　冒泡排序和简单选择排序 …… 45
2.8.3　Floyd 最短路径算法 …… 46
2.9　习题 …… 48

第 3 章　函数 …… 53
3.1　定义函数、调用函数、参数传递 …… 53
3.1.1　定义函数和调用函数 …… 53
3.1.2　参数传递 …… 54
3.1.3　return 语句 …… 56

電子工業出版 PUBLISHING HOUSE OF ELECTRONICS I

3.1.4 文档字符串 …… 57
3.2 全局变量和局部变量 …… 58
3.2.1 全局变量 …… 58
3.2.2 局部变量 …… 59
3.3 函数的参数 …… 60
3.3.1 默认形参 …… 60
3.3.2 位置实参和关键字实参 …… 61
3.3.3 任意形参(可变形参) …… 62
3.3.4 字典形参 …… 63
3.3.5 解封参数列表 …… 64
3.4 递归函数(调用自身的函数) …… 65
3.4.1 递归函数的使用方法 …… 65
3.4.2 实战：二分查找的递归实现 …… 67
3.4.3 实战：汉诺塔问题 …… 67
3.4.4 实战：快速排序算法 …… 68
3.4.5 实战：迷宫问题 …… 70
3.5 函数对象和 lambda 表达式 …… 72
3.5.1 函数对象 …… 72
3.5.2 lambda 表达式 …… 75
3.6 模块和包 …… 78
3.6.1 模块 …… 78
3.6.2 sys 模块(Python 解释器接口) … 83
3.6.3 伪随机数发生器模块 …… 86
3.6.4 包 …… 88
3.6.5 Matplotlib 包 …… 92
3.7 实战：Pong 游戏 …… 95
3.7.1 Pygame 游戏库介绍 …… 95
3.7.2 用 Pygame 编写游戏 …… 96
3.7.3 Pong 游戏 …… 98
3.8 实战：线性回归 …… 103
3.8.1 机器学习 …… 103
3.8.2 假设函数、回归和分类 …… 104
3.8.3 线性回归 …… 105
3.8.4 多变量函数的最小值、正规方程 …… 106
3.8.5 梯度下降法 …… 107
3.8.6 梯度下降法求解线性回归问题：模拟数据 …… 108
3.8.7 批梯度下降法 …… 112
3.8.8 房屋价格预测 …… 114
3.8.9 样本特征的规范化 …… 114
3.8.10 利用预测模型预测房屋价格 … 116
3.9 习题 …… 117

第 4 章 内置数据类型 …… 119
4.1 数值 …… 119
4.1.1 int、float、complex、bool …… 119
4.1.2 类型转换 …… 120
4.1.3 浮点数的精度问题 …… 121
4.1.4 数值计算的函数 …… 122
4.1.5 数学模块 …… 122
4.2 列表 …… 126
4.2.1 列表的定义 …… 126
4.2.2 访问 list 的元素(索引和切片) …… 127
4.2.3 包含和遍历 …… 128
4.2.4 list 的算术运算 …… 129
4.2.5 Python 的内置函数对 list 进行操作 …… 129
4.2.6 list 的方法 …… 131
4.2.7 列表解析式 …… 133
4.2.8 list 包含的不是对象本身而是对象的引用 …… 133
4.3 字符串 …… 134
4.3.1 定义字符串 …… 134
4.3.2 转义字符 …… 136
4.3.3 索引 …… 137
4.3.4 切片 …… 138
4.3.5 字符串不可修改 …… 138
4.3.6 包含和遍历 …… 139
4.3.7 内置函数对字符串操作 …… 139
4.3.8 字符串的方法 …… 140
4.4 元组 …… 144
4.4.1 创建 tuple 对象 …… 144
4.4.2 索引和切片 …… 146
4.4.3 tuple 是不可变的 …… 146
4.4.4 用内置函数对 tuple 操作 …… 148
4.4.5 tuple 的方法 …… 148
4.5 集合 …… 149
4.5.1 创建 set 对象 …… 149
4.5.2 遍历 set …… 151
4.5.3 用内置函数对 set 操作 …… 151
4.5.4 set 的方法 …… 151
4.5.5 set 的运算符操作 …… 153
4.5.6 set 的集合运算(并、交、对称差) …… 153
4.6 字典 …… 154

電子工業出版
PUBLISHING HOUSE OF ELECTRONICS I

4.6.1 创建字典对象……155
4.6.2 获取键的值……156
4.6.3 通过下标插入或更新一个键值……156
4.6.4 插入或更新多个键值：update()方法……157
4.6.5 删除键值……157
4.6.6 获取所有键、所有值、所有键值……158
4.6.7 遍历所有键、所有值、所有键值……158
4.6.8 用内置函数访问 dict 对象……158
4.6.9 从两个可迭代对象创建一个 dict……159
4.6.10 用 in 检测 dict 对象是否包含某个键……159
4.7 用强化学习 Q-Learning 算法求解最佳路径……159
4.7.1 强化学习……159
4.7.2 Q-Learning 算法……161
4.7.3 Q-Learning 算法的 Python 实现……162
4.8 习题……167

第 5 章 面向对象编程……173
5.1 什么是面向对象编程……173
5.1.1 过程式编程和面向对象编程…173
5.1.2 Python 既支持面向对象编程，也支持过程式编程……174
5.1.3 打印员工信息……175
5.2 类和对象……177
5.2.1 定义类……177
5.2.2 实例属性和构造函数……178
5.2.3 实例方法……180
5.2.4 类属性……181
5.2.5 del……183
5.2.6 访问控制和私有属性……184
5.2.7 运算符重载……186
5.3 派生类……187
5.3.1 派生类……187
5.3.2 覆盖……190
5.3.3 多继承……191
5.3.4 属性解析……192
5.4 绑定属性……193
5.4.1 动态绑定：给类和对象任意绑定属性……193
5.4.2 对象的__dict__属性……195
5.4.3 __slots__……195
5.5 实战：二叉搜索树……197
5.5.1 树、二叉树、二叉搜索树……197
5.5.2 树和二叉树的存储表示……199
5.5.3 二叉树的操作……201
5.5.4 二叉搜索树的操作……202
5.6 实战：面向对象游戏引擎和仿“雷电战机”游戏……204
5.6.1 面向对象游戏引擎……205
5.6.2 Pong 游戏……209
5.6.3 仿“雷电战机”游戏……212
5.7 习题……224

第 6 章 输入/输出……228
6.1 标准输入/输出……228
6.1.1 标准输出函数 print()……228
6.1.2 格式化输出……228
6.1.3 美观输出函数 pprint()……230
6.1.4 标准输入(内置函数 input())…233
6.2 文件读/写……234
6.2.1 内置函数 open()……234
6.2.2 文件对象的方法……235
6.2.3 二进制文件读/写……238
6.2.4 tell()方法和 seek()方法……239
6.3 习题……239

第 7 章 错误和异常……241
7.1 错误……241
7.1.1 语法错误……241
7.1.2 运行时错误：异常……242
7.1.3 逻辑错误……243
7.2 异常处理……244
7.2.1 捕捉异常的基本形式……244
7.2.2 捕获特定类型的异常……245
7.2.3 捕获未知的内置异常……245
7.2.4 else 子句……246
7.2.5 finally 子句……247
7.2.6 用 raise 抛出异常……248
7.2.7 自定义异常类……248
7.2.8 预定义清理行为……249

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

7.3 调试程序 …… 249
7.3.1 输出(打印) …… 249
7.3.2 断言 …… 250
7.3.3 日志 …… 251
7.3.4 调试工具 …… 252
7.4 习题 …… 253

第 8 章 高级语法特性 …… 256
8.1 容器、可迭代对象、迭代器、生成器 …… 256
8.1.1 容器 …… 256
8.1.2 可迭代的和迭代器 …… 256
8.1.3 生成器 …… 261
8.1.4 例子：读取多个文件 …… 264
8.1.5 标准库的迭代器工具 …… 265
8.2 闭包 …… 268
8.2.1 作用域 …… 268
8.2.2 嵌套函数 …… 269
8.2.3 什么是闭包 …… 270
8.2.4 用闭包代替类 …… 270
8.2.5 函数的闭包属性__closure__ … 271
8.3 装饰器 …… 272
8.4 @property …… 278
8.5 类的静态方法和类方法 …… 282
8.5.1 静态方法 …… 282
8.5.2 类方法 …… 282
8.6 浅拷贝、深拷贝 …… 284
8.6.1 浅拷贝 …… 285
8.6.2 深拷贝 …… 287
8.7 习题 …… 288

下篇 Python 标准库

第 9 章 标准库的常用模块 …… 293
9.1 操作系统接口模块 …… 293
9.1.1 os 模块 …… 293
9.1.2 高层文件操作 …… 296
9.1.3 glob 模块 …… 301
9.2 时间和日期模块 …… 302
9.2.1 时间模块 …… 302
9.2.2 日期模块 …… 303
9.3 习题 …… 306

第 10 章 正则表达式 …… 307
10.1 正则表达式的定义 …… 307
10.2 re 模块 …… 308
10.2.1 re 模块的常用函数 …… 309
10.2.2 编译模式串 …… 309
10.2.3 从头匹配 …… 310
10.2.4 多个匹配 …… 310
10.2.5 按匹配切分 …… 311
10.2.6 替换匹配 …… 312
10.3 正则表达式中的语法规则 …… 312
10.3.1 字符集 …… 313
10.3.2 反斜杠 …… 313
10.3.3 量词(重复) …… 314
10.3.4 边界字符(锚点) …… 314
10.3.5 或运算 …… 315
10.3.6 分组 …… 315
10.4 match 和 flags …… 317
10.4.1 match 对象及其应用 …… 317
10.4.2 标志参数 …… 319
10.5 习题 …… 320

第 11 章 并发计算 …… 321
11.1 多线程 …… 321
11.1.1 Thread 类 …… 321
11.1.2 线程同步 …… 327
11.2 多进程 …… 333
11.2.1 创建进程 …… 333
11.2.2 从 Process 类派生自己的进程类 …… 334
11.2.3 为进程命名 …… 334

第 12 章 图形用户接口(GUI)编程 …… 336
12.1 Tkinter 基础 …… 336
12.1.1 事件驱动编程 …… 336
12.1.2 第一个 GUI 程序 …… 336
12.1.3 Tkinter 部件 …… 337
12.1.4 布局——几何管理 …… 337
12.1.5 属性 …… 339
12.1.6 自定义事件处理函数 …… 340
12.1.7 定制事件处理函数 …… 340
12.1.8 文本输入框 …… 342
12.1.9 获取焦点 …… 343
12.1.10 聊天对话框 …… 343
12.1.11 框架 …… 344
12.2 用类封装 GUI …… 347

電子工業出版 PUBLISHING HOUSE OF ELECTRONICS I

12.2.1 菜单 …… 347
12.2.2 工具条 …… 351
12.2.3 画图 …… 352
12.2.4 用鼠标画图 …… 354

第 13 章 网络套接字编程 …… 356
13.1 套接字编程概述 …… 356
13.1.1 创建一个 socket 对象 …… 357
13.1.2 服务器：绑定地址 …… 357
13.1.3 面向连接的监听 …… 357
13.1.4 发送和接收数据 …… 358
13.2 TCP 服务器程序和客户程序 …… 358
13.2.1 最简单的 TCP 服务器程序和客户程序 …… 358
13.2.2 TCP 服务器程序和客户程序(多连接) …… 360
13.2.3 TCP 服务器程序和客户程序(数据分块) …… 362
13.2.4 TCP 服务器程序(多进程) … 364
13.2.5 TCP 服务器程序(多线程) … 365
13.3 UDP 服务器程序和客户程序 …… 367
13.3.1 UDP 服务器程序 …… 367
13.3.2 UDP 客户程序 …… 368
13.4 socketserver …… 369
13.4.1 socketserver 模块 …… 369
13.4.2 socketserver.TCPServer …… 369
13.4.3 socketserver.UDPServer …… 370

第 14 章 Internet 应用编程 …… 372
14.1 urllib 模块 …… 372
14.1.1 Get 请求 …… 372
14.1.2 Post 请求 …… 373
14.1.3 Request 对象 …… 374
14.1.4 代理服务器 …… 375
14.1.5 登录验证 …… 376
14.1.6 网络爬虫 …… 376
14.2 email …… 377
14.2.1 smtplib 模块 …… 377
14.2.2 收取和处理邮件 …… 380

第 15 章 数据持久化 …… 388
15.1 pickle 模块 …… 388
15.2 shelve 模块 …… 391
15.3 dbm 模块 …… 392
15.4 json 模块 …… 393
15.4.1 简单数据类型的编码和解码 …… 394
15.4.2 自定义类型的编码和解码 … 395
15.4.3 编码类和解码类 …… 397
15.4.4 流或文件 …… 398
15.5 sqlite3 模块 …… 399
15.5.1 数据库基本操作 …… 400
15.5.2 在查询中使用变量 …… 406
15.5.3 事务 …… 409

参考文献 …… 415

電子工業出版 PUBLISHING HOUSE OF ELECTRONICS I

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Python 语法与实践

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 1 章　Python 介绍

1.1　程序与编程语言

1.1.1　计算机是什么

计算机是一种根据指令对数据进行处理的通用计算设备。每台计算机都有一个称为“中央处理单元(CPU)”的微处理器芯片用于执行对数据进行处理的指令，不同计算机的指令集是不一样的。

1．计算机指令

计算机接收一系列指令作为输入，然后逐个处理它们，最后输出某些信息以显示它已完成的操作。这一过程类似人们日常生活中通过一系列操作步骤完成一个任务的过程。例如，一个人通过下列步骤完成“做饭”的任务：

```
从容器(米桶)中取出米，放入洗米盆；
用水对洗米盆中的米进行冲洗；
如果电饭煲没洗净
    清空并洗净电饭煲；
打开电饭煲的盖子，将米和水放入电饭煲；
插上电源，按下开关；
饭做好后，拔下电源(任务结束)。
```

虽然人们可以理解自然语言(如英语)中的复杂指令，但计算机只能理解用计算机语言表达的非常简单的机器指令集中的指令。无论多么复杂的计算，在计算机内都会被分解成许多条简单的可逐条执行的机器指令。告诉计算机如何执行复杂任务的指令序列称为**程序**。

以下是一些简单的计算机指令示例。

- 算术：加、减、乘或除。执行这些指令的操作称为**算术操作**(运算)。
- 比较：比较两个数字，查看哪个值更大，或者它们是否相等。执行这些指令的操作称为**逻辑操作**(运算)。
- 分支：跳转到程序的其他指令处，并从那里继续运行程序。执行这些操作的指令称为**控制语句**。

2．计算机的组成部分

计算机包含以下四种主要类型的组件或设备。

- 输入设备：允许计算机从用户处接收信息的设备，包括键盘、鼠标、扫描仪和麦克风。
- 处理组件：处理信息的计算机组件。计算机的主要处理组件是中央处理单元(CPU)，但在现代计算机中也可能有其他处理单元。例如，许多图形卡都带有图形处理单元(GPU)，GPU 以前只用于处理图形，但现在也可用于处理通用程序。
- 存储组件：存储信息的组件，包括主存储器(也称“**内存**”)和二级存储器(如硬盘驱动器、CD 或闪存盘等外部存储器)。存储组件是存储程序的指令和数据的地方。
- 输出设备：用于向用户显示信息的任何设备，包括显示器、扬声器和打印机。

可以用自动售票机来理解计算机的组件或设备(尽管自动售票机严格地讲并不是计算机)。

- 输入设备：投币口和选择按钮是自动售票机的输入设备。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- 处理组件：当使用者进行选择时，自动售票机执行的操作包括验证是否有满足条件的票；验证身份信息；检查和验证是否收到足够的资金；修改数据库；计算差额。执行所有这些操作的机器部分可以看作处理组件。
- 输出设备：显示结果；打印票据；退回多余资金。
- 存储组件：保存销售数据及价格等信息。

3．中央处理单元(CPU)

CPU 是计算机中最重要的部分，是计算机的“大脑”，主要负责计算、处理数据、控制其他设备等工作。它有以下几个重要的子组件。

- 算术/逻辑单元(ALU)：执行算术和比较运算。
- 控制单元：确定下一个要执行的指令。
- 寄存器：形成一个高速存储区以保存临时的运行结果。

不同种类的 CPU 可以处理不同的指令集，如 Intel IA-32、x86-64、IBM PowerPC 或 ARM 等。

4．存储器(Memory)

计算机将信息(程序、数据)存储在存储器中，存储器分为两类，**主存储器**(也称**内存**)和**辅助存储器**(也称**外存**)。

主存储器直接连接 CPU(或其他处理单元)，通常称为**随机存取存储器(RAM)**。计算机关闭时，大多数主存储器都会丢失其内容，即具有“易失性”。

可以将主存储器看作一组一排乘一列的存储器单元，每个存储器单元都可以通过其存储器地址寻址。第一个单元的地址为零，并且每个后续单元的地址比它之前的地址多一个，正如一个班级的学生的学号从 1 开始依次递增一样。每个存储器单元只能保存长度固定的用二进制表示的数值，但 CPU 可以随时用新的数值替换原有数值。

辅助存储器比主存储器价格便宜，但可以存储更多内容。虽然辅助存储器的存储速度比主存储器慢得多，但它是非易失性的，也就是说，即使在计算机关闭后其保存的信息也会保留，如硬盘和闪存盘。

计算机的操作系统提供操作辅助存储器的高级接口，这些接口允许信息以文件的形式保存在辅助存储器中，并且文件组织呈目录式的层级结构。接口和层级结构通常称为“文件系统”。

1.1.2 计算机编程

1．算法(Algorithms)

算法是完成某个任务或解决某个问题的一系列步骤(指令)。

2．编程和程序

编程就是使用计算机的指令来表示算法，即将算法转换为计算机可以执行的程序。程序就是算法在计算机中的表示和实现。

3．二进制

因为计算机硬件是由很多晶体管元器件组成的，而晶体管元器件只有“开”和“关”两种状态，所以 1 个晶体管只能表示两个数字：0 和 1。通过多个表示 0 或 1 的晶体管元器件可以组合出更复杂的数值，如整数或字符等。无论多么复杂的程序数据，在计算机硬件中都是以二进制(0 和 1)的形式表示的。

1 个晶体管元器件只能表示 1 位二进制数(0 或 1)，称为 1 **比特**(**Bit**)，简记为 b 或 1 位。8 个晶体管元器件可以表示 8 位二进制数字，即可表示 2^8 个不同的数值。8 位二进制数，称为 1 **字节**(**Byte**，简记为 B)。16 个晶体管元器件可以表示 16 位二进制数，即 2B。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
8×1024 个晶体管元器件可以表示 1024B，即 1KB。
8×1024×1024 个晶体管元器件可以表示 1024KB，即 1MB。
8×1024×1024×1024 个晶体管元器件可以表示 1024MB，即 1GB。
```

4．机器语言(machine language)

计算机中的指令和数据都是用 0 和 1 表示的。机器语言是用这种二进制数表示的、计算机能够直接识别和执行的机器指令集合。

例如，下列是将 17 和 20 相加的机器指令(采用 Intel 8086 机器语言，Intel Pentium 机器语言的子集)：

```
1011 0000 0001 0001
0000 0100 0001 0100
1010 0010 0100 1000 0000 0000
```

第一行告诉计算机将 17 复制到 AL 寄存器：前 4 位二进制数(1011)告诉计算机将信息复制到寄存器中，接下来的 4 位二进制数(0000)告诉计算机使用名为 AL 的寄存器，最后 8 位二进制数(0001 0001，表示整数 17)为要复制的数。

用机器语言编写程序非常困难，也很难被人们阅读和理解，但在 20 世纪 40 年代，第一台计算机的程序员必须这样做，他们通过纸上打孔表示 0 和 1 来编写机器语言的程序，因为没有其他选择！

5．汇编语言(assembly language)

为了简化编程过程，人们引入了汇编语言。每条汇编指令对应一条机器语言指令，使人们更容易理解，如上述的机器语言用 8086 汇编语言中等效的加法程序可写为：

```
MOV AL, 17D
ADD AL, 20D
MOV [SUM], AL
```

用汇编语言编写的程序不能直接被计算机理解，因此需要翻译步骤。一种叫作“**汇编程序**”的程序可以将汇编语言编写的程序转化为机器语言程序。

6．高级语言(High-level language)

虽然汇编语言对机器语言有很大的改进，但它仍然很神秘，而且它的级别太低，和机器语言一样，即便是完成最简单的任务也需要很多条指令。于是，人们发明了高级语言，使编程变得更加容易。

在高级语言中，一条指令可以对应多条机器语言指令，高级语言采用类人类的语言表示指令，这使得程序更易于读取和写入。以下是上述代码的 Python 等效代码：

```
sum = 17 + 20
```

1.1.3 编译器、解释器和 Python 语言

用高级语言编写的程序，在计算机执行之前也必须翻译为机器语言。某些编程语言的程序首先被整体翻译为机器语言的程序，并存储在指定文件中，然后再执行，这种语言称为**编译型语言**。也某些语言的程序语句被逐行翻译后再逐行执行，这种语言被称为**解释型语言**。Python 就是解释型语言。

编译型语言通过一个**编译器**程序，将编译型语言编写的源文件编译为可执行的二进制程序文件。解释型语言通过一个**解释器**程序，对解释型语言编写的源文件的每一条语句逐条解释并执行。

- **编译器**：编译器是将整个源代码程序一次性全部转换为机器语言代码的工具。转化后的机器语言代码可以直接在计算机上运行。
- **解释器**：是“一行一行”地解释执行，即将每条语句翻译成解释器自己的中间代码，然后再由解释器执行这个中间代码。每条语句是在解释器自己的虚拟环境中执行的。

解释器对每条语句逐条转换执行，使初学者很容易知道程序的错误位置。而编译器对整个源程

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

序进行一次性的转换，其优点是可以对代码进行整体的优化，从而提高程序的性能。

要求运行速度高或需要直接操纵硬件的程序通常使用 C 或 C++语言编写。C 或 C++语言是编译型语言，编译器将 C 或 C++语言程序编译为机器指令时，会对程序做很多细粒度的控制和优化，从而可以提高程序的运行速度，但编译需要耗费时间并且容易出错，而且需要整个程序写完且编译好后才能开始执行程序，而 Python 作为一种解释型语言，每条语句被逐条解释执行，可以立即发现程序错误，并可以看到程序运行结果，因此学习和编写这种解释型语言更容易。

1.1.4 Python 程序开发步骤

利用 Python 编写程序要经历以下几个步骤。

(1)理解问题：如这是一个什么样的问题？输入数据是什么？要产生什么结果？

(2)提出算法：解决这个问题的指令(步骤)序列。

(3)编写程序：将算法转换成某种编程语言的程序。

(4)测试：输入各种可能的不同的数据，检测是否产生预期的结果。

例如，要计算一组数值的平均值，可以按照上述步骤进行。

(1)理解问题：这些数值从哪里输入(键盘还是文件)？结果如何显示(屏幕打印输出还是保存到文件)？

(2)提出算法：首先用两个数值分别表示总和(sum)及数值的个数(计数器，count)，然后将输入的数值累加到总和上，同时累计数值的个数，最后用总和除数值的个数，得到平均值。人们经常以一种“伪代码”的方式描述算法的过程：

```
-----------start-------------
总和 sum=0。
计数器为 count=0。
重复：
    读一个值，
    如果读取值失败，结束这个“重复”过程，
    否则：
        将读取的值 value 加到 sum。
        计数器增加 1。

通过“总和”及“计算器”相除得到平均值。
显示/打印平均值。
-----------end------------
```

(3)编写程序：将算法用 Python 语言表示出来。

(4)测试： 输入不同的测试数据，查看结果是否正确。输入的数据可以包含非法数据，查看程序能否适当地应对。例如，输入的数据是字符串而不是数值，程序是否会提示等。

1.2 Python 语言的主要特征

Python 语言是由 Guido Van Rossum 于 1991 年发明的高级通用编程语言。所谓高级，是指它隐藏了机器指令的底层细节，并且提供给程序员的是一种人类易于理解的编程语言的语法。所谓通用，是指它可以开发各类应用程序。和其他高级编程语言，如 C、C++、Java 等不同，Python 语言以其简单易懂和编程效率高而著名。

Python 语言的主要特征如下。

1．简单易学

Python 语言简单，特别适合初学者学习和使用，使初学者可以把精力集中在问题本身和求解方

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

法上，而不用担心语法、类型等外在因素。Python 语言对代码的逐句解释执行方式，可以帮助初学者非常容易发现程序错误。正是由于其简单性，国内外很多教育机构都将 Python 语言作为中小学的首选编程语言。

2．编程效率高

同一个任务，用 Python 语言编写的代码往往是用其他编程语言所编写的代码的代码量的几分之一甚至几十分之一。例如，有时，用其他编程语言可能需要编写几百行甚至几千行代码，用 Python 可能只需编写几行代码。

3．跨平台

Python 语言是一个跨平台的编程语言，用该语言编写的程序可以在各种操作系统，如 Windows、UNIX/Linux、Mac，甚至手机操作系统，如 Android 等，平台上运行。

4．解释性语言

Python 语言是一个解释型编程语言，而 C、C++则是编译型编程语言。

5．大量的库

Python 语言提供大量丰富的库，包括 Python 语言自带的标准库和第三方的库。利用这些大量优秀的库，可以避免程序员重复“造轮子”。

由于 Python 语言的这些优点，Python 语言不再是程序员的专利，各行各业的人都在使用 Python 语言处理他们的数据和业务。特别是处理大数据和人工智能，都普遍采用 Python 语言作为其编程开发的语言。目前，世界范围内掀起了学习 Python 语言的热潮，从计算机专业到其他专业的研究开发人员，从大学生到中小学生，都在学习 Python 语言，国内许多大学、中学、小学都已经将 Python 语言作为首选编程语言。教育部考试中心于 2017 年 10 月 11 日发布了《关于全国计算机等级（NCRE）体系调整》的通知，决定自 2018 年 3 月起，在计算机二级考试加入“Python 语言程序设计”科目。

在 2017 年 IEEE 编程语言排行榜中，Python 语言排名第一。

1.3 Python 开发环境及安装方式

1.3.1 安装 Python

Python 开发环境的安装方式通常有两种方式，原生安装和工具包安装。

1．原生安装

原生安装是指只安装相应平台最基本的 Python 解释器。在 https:// www.python.org/downloads 下载 Python 安装程序并运行即可。

如图 1-1 所示，在安装过程中勾选“Add Python 3.6 to PATH”，安装程序会自动将 Python 解释器的路径添加到系统路径。

安装完成后，可以打开终端窗口，在其中输入“python”就可以打开 Python 解释器，系统将显示 Python 的版本信息：

```
>>> python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit
        (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
```

原生安装只安装最基本的 Python 解释器和一些自带的 Python 库。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

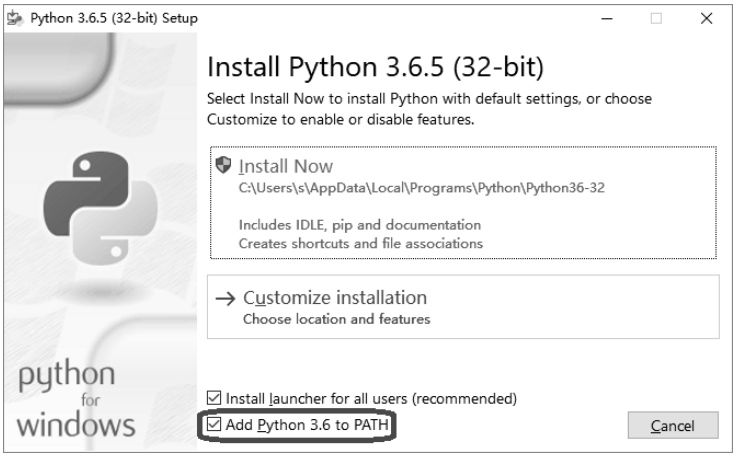


图 1-1　Python 原生安装

2. 工具包安装

一些安装工具包中，如 Anaconda 等，除包含 Python 官方的 Python 解释器及库外，还包含许多常用的第三方 Python 库，极大地方便了 Python 及第三方库的安装。

上述安装过程都是在 Windows 环境，如果在 UNIX/Linux 或 Mac 环境，则请读者自行在网上搜索安装方法。

1.3.2　Python 开发环境

可以在 Python 命令行直接编写交互式 Python 程序，也可以用操作系统自带的文本编辑工具(如 Windows 的记事本、UNIX/Linux 上的 vi 或 vim)或流行的文本编辑工具(如 sublime)编写 Python 程序，然后在 Python 的命令行运行这些 Python 程序。

但程序员更多采用以下 Python 编程环境。

- 将流行的文本编辑器，如 sublime、atom、Visual Code 等，进行简单的配置，使其可以编辑代码，然后直接在编辑器中调用 Python 解释器执行 Python 程序。读者可以自行在网络上搜索具体配置过程。
- 使用集成开发环境，如 Pycharm，进行 Python 程序的开发。
- 使用 Jupyter Notebook。

Jupyter Notebook 可以将浏览器当作 Python 程序的编辑工具，不仅可以在其中编写代码，还可以使用丰富的格式单元(markdown)编写丰富的文字、公式等，从而将代码和文章有机地结合在一个文档中。因此，Jupyter Notebook 是 Python 程序员使用最多的用于代码编写和记录思想的工具。Anaconda 安装程序已经自带了 Jupyter Notebook 工具。如果是原生安装的 Python，则只要在 Python 提示符下输入：

```
pip install --upgrade pip
pip install jupyter
```

本书配套的视频教程采用 Jupyter Notebook 作为教学环境。

1.4　Python 解释器

Python 解释器用于运行 Python 命令(程序)，主要有两种模式，交互式解释执行和运行脚本文件。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

1.4.1 交互式解释执行模式

1．交互式解释执行模式应用方法

只要在命令行输入“python”就可进入交互解释执行模式。在此模式下，在 Python 解释器的提示符>>>后可以直接输入 Python 的指令（也称“语句”），解释器将会解释执行这条语句。例如，输入 1+1：

```
>>> 1+1
2
>>>
```

屏幕窗口立即输出了计算结果 2。

接着再输入 50 – 5/6：

```
>>> 50 - 5/6
49.166666666666664
```

继续输入"hello world!"：

```
>>>"hello world!"
'hello world!'
```

这里我们输入了用双引号括起来的一串字符，输出的是以双引号或单引号括起来的同样的一串字符。这种以单引号或双引号括起来的字符序列叫作“**字符串**”。

再继续输入下列语句：

```
>>> print("hello world!")
hello world!
```

屏幕窗口将输出一串字符 hello world!。

2．如何退出 Python 解释器

在 Windows 平台输入【Ctrl+Z】，在 Linux、UNIX 或 Mac 平台输入【Ctrl+D】，将退出 Python 解释器。

1.4.2 函数

上例代码调用了 Python 的一个叫作 print()的内置函数，用于在显示窗口输出信息，将双引号括起的一个字符串“hello world!”传给 print，即放在 print 后面的圆括号()里，函数 print()将在显示窗口输出字符的内容。我们习惯称 print()为打印（或输出）函数。

注意

函数 print()输出的字符串没有包括字符串的开始和结束位置的单引号或双引号，开始和结束位置的单引号或双引号只是用来表示一个字符串的，它们本身并不是字符串的内容。

需要说明的是，Python 的函数并不是 Python 语言的语法组成部分，**函数**实际上是一组语句的名字，程序员可以用 Python 语句编写各种各样的函数，每个函数完成一个特定的专门的工作。Python 语言的实现者除提供了一个执行 Python 语句的解释器外，通常会也提供一些最常使用的函数，它们被称为“**内置函数**”。程序员也可以定义自己的函数，它们和内置函数在定义和使用上没有区别。通过将一组排列好的指令语句用函数名命名，就可以多次通过这个**函数名**去调用这个函数名指示的这组指令语句（也称**代码块**或**程序块**），从而可以避免多次重复编写同样的指令语句，极大地提高了编程效率。

定义每个函数时，都会说明该函数可以接收什么样的数据。在调用该函数时，应该按照该函数的规范传递合适的数据给这个被调用的函数。这些传递给函数的数据称为“**实际参数**”，这些“实际参数”都是放在函数名后的圆括号里传递给函数的。例如，print("hello world!")就是通过圆括号传递给函数 print()一个双引号括起来的字符串。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

如果字符串没有用双引号或单引号括起来，则会出错。例如：

```
print(hello)
--------------
NameError     Traceback (most recent call last)
<ipython-input-6-1cd80308eb4c> in <module>()
----> 1 print(hello)
NameError: name 'hello' is not defined
```

同样，调用函数时，如果参数不是放在函数名后的圆括号里传递给该函数，则也会出错：

```
print"hello"
--------------
File "<ipython-input-7-26542d529988>", line 1
   print"hello"

SyntaxError: invalid syntax
```

再看下面的例子：

```
>>> print("1+2")
1+2
>>> print('1+2')
1+2
```


两个函数 print()输出的都是同样的字符串("1+2"和'1+2')，但是，如果输入 print(1+2)，则：

```
>>> print(1+2)
3
```

输出的结果是 3 而不是字符串。

Python 的语句一般写在一行里，如果一条语句要写在多行，则可以在每行的结束位置添加一个特殊的斜杠符号\，则表示 Python 语句将继续包含后面的行。例如：

```
>>>1+\
>>>2
3
```

如果写成下面的形式，则会出错：

```
>>>1+
>>>2
File "<ipython-input-9-c5120ce51ab9>", line 1
   1+

SyntaxError: invalid syntax
```

如果字符串中内容要写在多行，则也需要添加斜杠符号\。例如：

```
>>>print("hello,\
>>>world"\
>>>    )
hello,world
```

函数 print()可以有多个输出项，这些输出项在输出时，输出项之间以空格隔开。

```
>>>print('hello', 3.14, 2)
hello 3.14 2
```

函数 print()默认输出后换行，可以通过设置关键字参数 end 的值改变其行为。例如：

```
>>>print(1,end = ' ')
>>>print(2,end = ' ')
>>>print(3,end = ' ')
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
1 2 3
```

即在每个 print 语句的后面输出的是空格而不是换行。

1.4.3 运行脚本文件模式

可以将多个 Python 指令(命令)放在一个后缀为.py 的文本文件中，这种文件称为 **Python 脚本文件**(script)。例如，用文本编辑工具，如“记事本”，编辑下列代码：

```
print(1+2)
print("1+2")
print('1+2')
print("hello, world!")
print("hello", "world!")
```

并保存在命名为“first.py”的文件中。在命令行窗口输入：

```
>>>python first.py
```

Python 是解释器程序，因此解释器将解释执行 first.py 中的 Python 指令，并输出如下信息：

```
3
1+2
1+2
hello, world!
helloworld!
```

注：

- 函数 print()默认输出后换行。
- 函数 print()可以输出多个量，这些量以逗号隔开，如 print("hello", "world!")输出了两个字符串。
- 后缀为.py 的脚本文件也称为**模块**(**module**)。

1.4.4 语句和注释

Python 的命令如 1+2 或 print("hello,world")称为语句(也称**代码**)。

除语句外，在编写 Python 脚本文件时，我们经常会对程序添加一些说明或注解，这些说明或注解称为“**注释**”，它们本身并不是程序语句，其作用是为了帮助他人阅读或今后自己回顾程序。

添加注释的方法很简单，就是在一行文字的最前面加上一个特殊符号#，该行文字就成了注释。例如：

```
# 这是我的第一个 python 程序
# 程序功能是输出整数和字符串
print(1+1)
print('hello,world')  #打印字符串 hello,world
```

执行该脚本的结果：

```
2
hello,world
```

在每一行里，#后面的文字都是注释，如上例程序的第 1、2 行文字和第 4 行#后面的文字，都是程序的注释。删除注释对程序没有任何影响，但“为程序添加注释”是一个良好的编程习惯，不仅可以帮助他人阅读理解程序，也有助于自己今后回顾这些程序。

总结

- 编写 Python 程序的两种模式，交互式解释执行和运行脚本文件。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- 函数及函数调用的概念。
- 语句和注释的区别。

1.5 数和字符的表示

1.5.1 数的表示

1．十进制和二进制

程序的数据和代码在计算机内部都是以 0、1 表示的。而在编程语言中，数的表示方式有多种。例如，可以用日常生活中常用的“**十进制(Decimal)**”，即用 10 个不同的数字 0、1、2、...、9 表示一个数。对于小于 10 的数，可以直接用这 10 个不同数字中的一个来表示或区分就可以了，即 1 位数字就可以表示不超过 10 的数。但如果一个数超过了 10，就需要用两位数字、三位数字等多位数字来表示，即用“**逢十进一**”的多位数表示。例如，一个整数 329 意思是“3 个 100”加“2 个 10”加“1 个 9”，可以表示成 10^i 的多项式：

$$329 = 3 \times 100 + 2 \times 10 + 9 = 3 \times 10^2 + 2 \times 10^1 + 9 \times 10^0$$

但在计算机中，由于计算机硬件，即晶体管元器件，只有“开”和“关”两种状态，也就是在说，1 个晶体管元器件的开和关状态只能区分两种不同情况，相当于只能表示 0 和 1 两个数字。如果要表示更大的数字，则可以采用“**逢二进一**”的方法，即采用多个晶体管元器件，即多个 0 和 1 的排列来表示一个很大的数字。这种用两个数字 0 和 1 表示数的方法就称为“**二进制**”表示法。例如，二进制数 1011 可以表示成 2^i 的多项式：

$$1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 11$$

即相当于十进制数 11。

1 位二进制位经常也称为 1 比特(Bit，简记为 b)，而 8 位二进制位经常也称为 1 字节(Byte，简记为 B)。8 位二进制数对应的十进制数如表 1-1 所示。

使用前 7 位二进制位可以表示 0 到 127 共 128 个不同数字，使用全部 8 位二进制位可以表示一共 256，即 2^8 个不同的数。如果有 n 位二进制位，则可以表示 2^n 个不同的数，如正整数 0 到 2^n-1。

2．十六进制

当处理更大的二进制数时，则二进制数的位数太多不方便使用。例如：

```
1011 0101 1101 1001 1110 0101
```

如果表示成十进制数 11917797，则只要 8 位。然而，十进制数在某些情况下也不方便，如希望将从右往左的第 17 位二进制位设置成 1，就很难用十进制做到。解决方法就是采用“**十六进制**”，即用十六个不同的数字来表示一个数，即用十进制的 10 个数字 0、1、2、…、9 加上英文字母的前 6 个字母 A(a)、B(b)、C(c)、D(d)、E(e)、F(f)来分别表示一个数(英文字母不区分大小写)。如表 1-2 所示是十六进制、十进制和二进制之间的对应关系。

因为 1 位十六进制数对应 4 位二进制数，所以可以将任何 1 个二进制数按照 4 位一组的方式以十六进制形式表示。例如：

```
1011 0101 1101 1001 1110 0101
```

对应十六进制表示法，很简单就可以写出来：

```
b 5 d 9 e 5
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

对这种十六进制数可以很容易地设置其对应二进制的第 17 位为 1(思考一下?)，也可以将十六进制数采用如下方式计算出对应的十进制数的值：

$$b \times 16^5 + 5 \times 16^4 + d \times 16^3 + 9 \times 16^2 + e \times 16^1 + 5 \times 16^0$$

表 1-1　8 位二进制数对应的十进制数

二　进　制	十　进　制
0000 0000	0
0000 0001	1
0001 0001	17
0111 1111	127
1000 0000	128
1001 0001	145
1111 1111	255

表 1-2　十六进制、十进制和二进制的对应关系

十六进制	十　进　制	二　进　制
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A 或 a	10	1010
B 或 b	11	1011
C 或 c	12	1100
D 或 d	13	1101
E 或 e	14	1110
F 或 f	15	1111

1.5.2　字符的表示

在计算机中，各种字符，如英文字母、数字(0、1、2、...、9)、一些特殊字符(#、/、换行符、制表符等键盘上可见字符)都是以一串二进制数表示的。

1．ASCII 码

1960 年代，American Standard Code for Information Interchange（ASCII)约定使用 7 位二进制数表示 128 个不同的字符。

原始的 7 位 ASCII 对于英语没有任何问题，但是对于法语、德语等语言字符就不够用了，于是提出了扩展的用 8 位二进制数表示字符的扩展 ASCII 字符编码。

但对于某些国家语言的字符很多情况下(如汉字将近 88 000 个)，8 位 ASCII 字符编码就不够用了，为了克服原始的 ASCII 的缺陷，1990 年人们提出了“统一字符编码”(Universal Character Set, UCS)。UCS 是 ISO 10646 标准，UCS 字符的编码长度为 32 位。

2．UCS 和 Unicode

UCS 定义了字符和**编码点**(**Code Point**)的整数值的映射关系。编码点和**编码**(**Code**)并不是一回事，编码点是一个整数值，而编码是用一系列字节表示一个编码点的方式。当数值不超过 256 时编码点用一字节就可以表示，如果用 4 字节而不是 1 字节存储这些值就浪费了。因此，编码是有效存储编码点的方式。

Unicode 是一个标准，不仅定义了一个字符集及其字符的编码点(字符的编码点等同于 UCS 对应字符的编码点)，还定义了多种表示编码点的**编码方式**。编码点不仅可以表示所有语言的所有字符，还可以表示不同的图形符号，甚至表情符号。大多数语言字符都能用 16 位编码(Code)表示。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Unicode 提供了多种编码方法，但也造成了理解上的困惑，最广泛使用的 Unicode 编码方式包括 **UTF-8**、**UTF-16**、**UTF-32**。其中，每个编码方式都可以表示 Unicode 字符集中的所有字符。

- UTF-8 表示一个字符采用的是变长的字节序列(用 1～4 字节表示 1 个字符)，其中，ASCII 字符的编码使用 1 字节表示，和对应的 ASCII 编码是一样的。大多数网页的文本都采用 UTF-8 编码方式。
- UTF-16 用 1 个或 2 个 16 位(16-bit)编码表示一个字符。UTF-16 包含了 UTF-8。
- UTF-32 最简单，用 1 个 32 位(32-bit) 编码表示所有字符。

如表 1-3 所示是 UniCode 编码点和 UTF-8 编码的关系。

表 1-3　UniCode 编码点和 UTF-8 编码的关系

UniCode(十六进制形式)	UTF-8(二进制形式)
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

例如，汉字的“中”的十六进制表示是“4E2D”，其二进制形式是“0100 1110　0010 1101”，其 UTF-8 编码占用的是 3 字节，只要将其二进制位从后往前填入相应的“x”的位置就得到了其对应的 UTF-8 编码 1110**0100** 10**111000** 10**101101**。

如表 1-4 所示是字符“A”和汉字“中”的 UniCode 编码点及 UTF-8 编码。

表 1-4　字符 A 和汉字“中”的 UniCode 编码点及 UTF-8 编码

字符	ASCII	UniCode	UTF-8
A	01000001	00000000 01000001	01000001
中	无	01001110 00101101	11100100 10111000 10101101

1.6　如何获得帮助

学习 Python 语言(包括 Python 的标准库)时，可通过下面的一些途径寻求帮助。

- 使用 google 搜索，如在 www.google.com 中输入“python 函数”可以搜索到与 Python 函数相关的帮助文章。
- 阅读官方文档，如在 https://docs.python.org/3/查找需要的文档资料，虽然官方文档解释得不是很清楚，但仍然是最权威的官方解释。
- 在程序员问答网站，如 StackOverflow 和 Quora 查询问题答案，也可以提出问题或回答其他人的问题。
- 利用 Python 自带的“内省”功能(Python 自带的内省函数)，如利用函数 type()、dir()和 help()等查询某个模块函数或类的文档说明或属性。

1.7　习题

1. 分别以交互或解释执行模式和运行脚本文件模式实现以下问题。

(1)打印以下图案。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    *
  *   *
*   *   *
  *   *
    *
```

(2)计算半径是 2.5 厘米的圆的面积。

(3)输出以下语句：

```
这是我的第一个 Python 程序
This is my first Python grogram!
```

2. 写出对二进制数 17 和 32 进行加法运算的结果。
3. 将十进制整数 765431 转换为十六进制，在十六进制形式下，将该数的二进制的第 17 位设置为 1，并写出对应的二进制数。
4. 请在网络上查询换行符、回车符、制表符和汉字“汉”的 Unicode 编码点(值)，并根据表 1-4 写出其对应的 UTF-8 编码。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 2 章　Python 基本计算

2.1　值、类型、对象

2.1.1　值和类型

程序的数据是以值的形式体现的，而值有相应的数据类型。例如，2 是一个整数，3.14 是一个浮点实数，hello, world 是一个字符串。用 Python 的内置函数 type()可以查询一个值的数据类型。例如：

```
>>>type(2)
int
>>>type('hello,world')
str
>>>type(3.14)
float
```

可以看出，和其他编程语言一样，每个值都有确定的数据类型，以用于说明这种数据类型的值的内容是什么样的，占用多大的内存空间，取值范围是多少，能进行哪些运算(操作)。例如，int 类型的值可以是任意大的整数，只要计算机内存足够大，而其他编程语言，如 C 语言规定的 int 类型的值总是占用固定大小的内存(如 4 字节)。再如，Python 的 float 类型表示的浮点实数最多可以精确到 15 位小数。对于 int 或 float 类型的值可以用+、–、*、/、%、// 等运算符进行加、减、乘、除、求余数、整数除运算，但是整数除运算对于 float 类型的值则是转为整数类型进行的。例如：

```
print(5+2)            #加
print(2-5)            #减
print(5*2)            #乘
print(5/2)            #除,产生的是一个浮点数
print(5//2)           #运算符//表示整数除
print(5 ** 3)         #指数运算，计算 5 的 3 次方
print(3.14//2.5)      #整数除
print(3.14%2)         #求余数
print((50 - 5*6) / 4)
```

输出：

```
7
-3
10
2.5
2
125
1.0
1.1400000000000001
5.0
```

对字符串对象可以用+(加法运算)，但其含义是“拼接”的意思。例如：

```
print("hello," + "world")
```

输出：

```
hello,world
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

但对字符串对象不能进行减法、除法等运算。也就是说，不同类型的数据能够进行的运算不同。

```
print("hello," - "world")
print("hello," * "world")
print("hello," / "world")
```

产生 TypeError(类型错误)：

```
---------------------------------------------------------------------------
TypeError : Traceback (most recent call last)
<ipython-input-4-bb555d5919b3> in <module>()
----> 1 print("hello," - "world")
      2 print("hello," * "world")
      3 print("hello," / "world")
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

编译器提示：不能对两个字符串(str)类型的值进行–、/、*运算，但 1 个字符串可以和 1 个整数相乘，其结果是 1 个重复复制这个字符串的新字符串。例如：

```
print(3*"world")
print("world"*3)
```

输出：

```
worldworldworld
worldworldworld
```

注意

一个数据类型的值的表示形式并不完全和日常生活中的该类型的值的表示形式完全一致。例如日常生活中，整数 1000 经常也写成“1,000”，但在 Python 中不能用这种形式表示整数。执行下面语句：

```
>>>print(1,000)
```

输出的结果是：

```
1 0
```

这里的 1,000 在 Python 不是表示整数 1000，而是用逗号隔开的两个 int 类型整数，因此输出的是两个值。

2.1.2 对象

1. 对象

每个值都有具体内容和类型，要在计算机中表示这个值，就要给这个值分配一块单独的存储空间。因此，在 Python 中要用对象来刻画有独立内存空间和数据类型的值。也就是说，在 Python 中，程序中的数据都是以对象的形式存在的。

一个对象包含一个具体数据的三个特征，**值**、**数据类型**和 **id**。每个对象都有唯一的 id(相当于一个人的身份证号码)作为该对象的内存地址。可以用 Python 的内置函数 id()来查询一个对象的 id。例如：

```
print(id(2))
print(id(3.14))
print(id('hello,world'))
```

输出：

```
1854893568
1775344243768
1775345251504
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

可以看出，这三个对象都有自己唯一的 id(各自独立的存储空间)。

2．空值(None)

Python 有一个特殊的值 None，称为“**空值**”(**None**)，用于表示一个没有值的对象。

除这些表示数据的**值对象**外，本书后续将介绍的函数、类等在 Python 中都是对象，即 **Python 的一切都是对象**。

总结

- 每个值都有对应的数据类型，如 3 是一个 int(整型)类型的值，3.14 是一个 float(浮点型)类型的值，hello world 是一个 str 类型(字符串类型)的值。
- 数据类型规定了这种类型的值的内容是什么、能进行什么样的操作(运算)、占用多大的内存空间，取值范围是多少。
- 值都是以对象的形式存在的，一个对象包含值、数据类型和 id。id 实际上就是对象的内存地址。可以用内置函数 type()和 id()查询一个对象的数据类型和 id。
- None 是一个没有值的空值对象。

2.2 变量

2.2.1 什么是变量

上述的 hello world、3.14、2 都是具体的值对象，也称为“**字面量**”。为了便于在程序代码中引用这些对象，可以给它们起名字，这些名字就称为“**变量**”，即可以定义一个变量引用这个对象。定义一个变量的格式是：

```
变量名 = 对象
```

变量名后跟一个等号=，等号后面是该变量引用的对象。定义变量的这个等号=也称**赋值运算符**，其真正含义不是给这个变量一个值，而是说明这个变量引用哪个对象，仅仅是这个对象的 1 个名字而已。例如：

```
#变量就是对象的名字，也称为对象的引用
PI = 3.14;
s = "hello world"
ival = 2;
print(ival)  #打印变量 ival 引用的对象 2
print(PI)
print(s)
```

输出：

```
2
3.14
hello world
```

上述代码分别给 3 个值对象 3.14、hello world 和 2 起了不同的名字 PI、s、ival，即定义了变量。

变量是**对象的引用**：变量本身并不存储具体数值，它们仅是对象的引用。可以用赋值运算符=修改变量引用的对象，使变量引用其他对象。例如：

```
PI=ival   # PI 修改为 ival 引用对象的引用，即 PI 和 ival 都引用了整数对象 2
print(PI,ival)
print(id(PI),id(ival))
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
2 2
1495887360 1495887360
```

因此，变量 PI 和 ival 都是同一个对象的 2 个不同名字而已，PI=ival 赋值前后如图 2-1 所示。

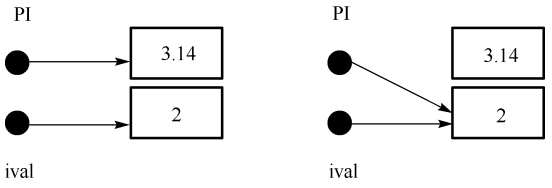


图 2-1　PI=ival 赋值前后

不能使用一个未定义的变量(未说明引用哪个对象)。例如：

```
print(radius)    # radius 是未定义的
```

将产生 NameError(名字错误)的语法错误：

```
NameError                 Traceback (most recent call last)
<ipython-input-3-d01a0099d01e> in <module>()
----> 1 print(radius)    # radius 是未定义的
NameError: name 'radius' is not defined
```

Python 中有两个运算符 == 和 is，前者用于比较对象的值是否相等，后者用于判断两个对象是否为同一个。例如：

```
a=1000
b=1000
print(a == b)
print(a is b)
```

输出：

```
True
False
```

True 和 **False** 是 Python 的 **bool**(布尔)类型的两个值，分别表示“真”和“假”。这里，a==b 比较的结果值是 True，即“真”，也就是说，a 和 b 的值是相等的。而 a is b 判断的结果值是 False，即“假”，也就是说，a 和 b 不是同一个对象。

可以首先用函数 type()得到 a == b 的类型，然后用函数 print()输出这个类型。例如：

```
print(type(a==b))
print(type(a is b))
```

输出：

```
<class 'bool'>
<class 'bool'>
```

可以用函数 id()得到 a、b 的内存地址。例如：

```
print(id(a))
print(id(b))
```

输出：

```
1524446335088
1524446335280
```

由此说明 a 和 b 确实是两个不同的对象，它们占据的内存块地址是不同的。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2.2.2 变量名和关键字

可以给变量起任意的名字，但为了方便阅读和理解程序，变量名应该尽量直观易懂。例如，应使用“PI”而不应用“sdfduzwlf2”来表示圆周率。Python 语言和其他编程语言一样，变量名只能包含字母、数字和下画线，且不能以数字开头。同时，变量名不能和已经被 Python 语言使用的名字(**关键字**)相同。例如，不能定义 **def** 为变量名，因为 **def** 是 Python 用于定义函数的关键字。

```
80ui = 23
A$2 = 11.2
class = "hello"
```

上面的三个变量名都是非法的(invalid)。其中，class 是 Python 用于定义类的关键字。

2.2.3 动态类型语言

在静态类型语言，如 C 语言中，定义变量时必须指明其数据类型，例如：

```
int i = 3;
```

数据类型规定了这种类型的变量占据多大的内存空间，数值的内容和取值范围，以及对这种类型的变量(数据)能进行什么样的运算。编译器会根据变量的类型为它分配适合的内存空间，检查其初始值是否合法，是否能进行某种运算。例如，编译器在编译下面的 C 语言代码时会报错：


```
int j = "hello";      //错：不能将一个字符串赋值给 int 整型变量
double a, b;
a = a%b;              //错：对于 double 类型的数据不能执行取余运算%
```

在静态类型语言中，变量是一块具有确定类型的内存的名字，一旦定义了该变量，则在销毁该变量前，其变量名始终指向这块内存。

与静态类型语言不同，Python 是一个动态类型语言，变量名仅是一个对象的名字，并不是占据一块内存的那个对象本身，一个变量名可以随时指向不同的对象，直到程序运行时，才能根据其指向的对象知道该对象的数据类型。例如：

```
s= "hw-dong"          #s 指向的对象"hw-dong"是一个 str 类型
s = 3.14              #s 指向的对象 3.14 是一个 float 类型
```

总结

- 通过赋值运算符=将一个对象赋值给一个变量名用于定义一个变量。
- 变量仅是对象的名字，或者说，变量引用了对象。一个对象既可以对应多个变量(名)，也可以随时用赋值运算符=使一个变量(名)指向另外的对象，即变量不会“从一而终”。
- 可以用等于运算符==判断两个对象的值是否相等，用 **is** 运算符判断两个对象是否为同一个对象。当然，也可以用函数 print()打印两个对象的 id，查看它们是否是同一个对象。
- 变量名只能包含字母、数字和下画线，且不能以数字开头，更不能与 Python 关键字相同。
- Python 是动态类型语言，只有运行时根据变量引用的对象才能知道这个对象的数据类型。

2.3 数据类型概述

Python 自带了一些**内在数据类型**(**内在类型**)，主要内在数据类型包括数值(number)、字符串(str)、列表(list)、元组(tuple)、集合(set)、字典(dict)。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2.3.1 数值类型

数值类型不是一个单独的类型，而可分为 int（整型）、float（浮点型）、complex（复数类型）、bool（布尔类型）。

bool 型用于表示一个逻辑量，即一个命题的真和假，因此只有表示真和假的两个值 True、False。

例如：

```
>>>type(5)
int
>>>type(3.14)
float
>>>type(2+3j)
complex
>>>type(False)
bool
```

只要计算机内存足够，Python 的整型的长度就可以任意长。但浮点型的值小数点后的精度不超过 15 位，而复数型的值总是写成 x+yi 形式。

```
a = 1234567890123456789
b = 0.1234567890123456789
c = 1+2j

print(a)
print(b)
print(c)
```

输出：

```
1234567890123456789
0.12345678901234568
(1+2j)
```

2.3.2 列表

list 是一个数据元素的有序序列，定义列表时用一对左右方括号[ ]包围，元素之间用逗号隔开。list 中的元素可以是不同的类型。例如：

```
[2, 3.14, 'python']
```

list 中的元素的类型是任意的，当然可以包含其他的 list 对象。例如：

```
[2, 3.14, [3,6,9],'python']
```

因为 list 是一个有序序列，所以其中的每个元素都有唯一的下标，下标从 0 开始，即第 1、2、3、…元素的下标依次是 0、1、2、…。因此，对于 list 对象可以用**下标运算符[]**访问它的一个元素或一系列连续的元素。

```
my_list =[2, 3.14, 8, 'python',9,'hello']
print(type(my_list))  #打印 my_list 的类型，即 list 类型
print(my_list)
print(my_list[0])
print(my_list[3])
```

输出：

```
<class 'list'>
[2, 3.14, 8, 'python', 9, 'hello']
2
python
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

还可以通过向**下标运算符[ ]**传递起始位置和结束位置的两个下标，返回连续多个元素组成的子列表(不包含结束位置的元素)。格式如下：

```
list[start:end]
```

起始位置和结束位置下标之间用冒号:隔开。例如：

```
my_list =[2, 3.14, 8, 'python',9,'hello']
print(my_list[1:4])  #从第 2 个到第 5 个，不包含第 5 个
print(my_list[2:])   #起始位置下标为 2，没有结束位置下标，表示从起始位置下标之后的所有元素
print(my_list[:4])   #没有起始位置下标，表示起始位置下标默认为 0
print(my_list[:])    #没有起始位置和结束位置下标，表示所有元素
```

输出：

```
[3.14, 8, 'python']
[8, 'python', 9, 'hello']
[2, 3.14, 8, 'python']
[2, 3.14, 8, 'python', 9, 'hello']
```

list 对象是**可修改的**(**mutable**)。例如：

```
my_list =[2, 3.14, 8, 'python',9,'hello']
my_list[2]='小白'
print(my_list)
```


输出：

```
[2, 3.14, '小白', 'python', 9, 'hello']
```

为指定下标范围内的元素重新赋值，从而替换为另一个 list。例如：

```
my_list =[2, 3.14, 8, 'python',9,'hello']
my_list[2:5]=[10,25]
print(my_list)
```

输出：

```
[2, 3.14, 10, 25, 'hello']
```

为指定下标范围内的元素赋值一个空的 list[ ]，相当于删除这个范围内的元素。例如：

```
my_list=[2,3.14,10,25,'hello']
my_list[2:4]=[]      #相当于删除了[2:4]之间的元素，不包括下标为 4 的元素
print(my_list)
```

输出：

```
[2, 3.14, 'hello']
```

必须给下标范围内的元素赋值一个 list 对象，如果赋值非 list 对象则会出错。例如：

```
my_list[1:3]= 5
```

产生 TypeError(类型错误)：

```
---------------------------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-10-22d1f85b1294> in <module>()
----> 1 my_list[1:3]= 5

TypeError: can only assign an iterable
```

2.3.3 元组

和 list 一样，元组(tuple)也是一个有序序列，也就是每个元素也有唯一的下标，但 tuple 是**不可修改的**(**immutable**)，另外，定义 tuple 时用圆括号而不是方括号。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
t = ('python',[2,5],37,3.14,"https://a.hwdong.com")
print(type(t))
```

输出：

```
<class 'tuple'>
```

可以用下标运算符[]访问 tuple 的元素：

```
print(t[2])
print(t[0:3])
print(t[:4])
```

输出：

```
37
('python', [2, 5], 37)
('python', [2, 5], 37, 3.14)
```

但不可以通过下标修改 tuple 对象，因为 tuple 是不可修改的。例如：

```
t[1] = 'helo'
```

产生下列类型错误(TypeError)：

```
TypeError                 Traceback (most recent call last)

<ipython-input-14-30808fa682a5> in <module>()
----> 1 t[1] = 'helo'
TypeError: 'tuple' object does not support item assignment
```


注意

只有一个元素的 tuple 的最后必须加一个逗号，如(25,)表示一个 tuple，而(25)表示一个整数。

2.3.4 字符串

1. 字符串(str)

字符串是 UniCode 字符的有序序列，可以用置于首尾的单引号或双引号包围一个字符序列来表示字符串。如果字符串的内容写在多行，则首尾分别要用三个单引号或双引号包围多行的字符序列。

单引号表示的字符串中可以包含双引号字符，但不能直接包含单引号(否则无法知道字符串的开始结尾在哪里)。同样，双引号表示的字符串中可以包含单引号，但不能直接包含双引号。例如：

```
print("教'小白'精通编程博客")
print('教"小白"精通编程博客')
print(''' '教小白精通编程博客'
https://a.hwdong.com
python 语言
''')
```

输出：

```
教'小白'精通编程博客
教"小白"精通编程博客
'教小白精通编程博客'
https://a.hwdong.com
python 语言
```

单引号表示的字符串中不能直接包含单引号，双引号的字符串中也不能直接包含双引号。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print('教'小白精通编程博客')    #单引号表示的字符串中不能包含单引号
```

产生如下语法错误：

```
File "<ipython-input-24-11082e72943d>", line 1
   print('教'小白精通编程博客')    #单引号表示的字符串中不能包含单引号
                 ^
SyntaxError: invalid syntax
```

和 tuple 一样，str 类型的对象也是**不可修改的(immutable)**。例如：

```
s  = 'hello world'
s[1] = 'E'
```

产生“字符串对象不支持赋值”的 TypeError(类型错误)：

```
---------------------------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-1-8616f9d54e50> in <module>()
     1 s  = 'hello world'
----> 2 s[1] = 'E'

TypeError: 'str' object does not support item assignment
```

2. 转义字符

如果要在单引号表示的字符串中包含单引号，则可以在其前面添加反斜杠符号\构成一个单独的字符。例如，用\'表示单引号字符，这种前面加反斜杠字符表示字符'的 \'称为**“转义字符”**。转义字符表示的是一个字符而不是两个字符。

```
print('hello \'li ping\'')
print(len('hello \'li ping\''))   #打印字符串的长度，\'表示的是一个字符
```

输出：

```
hello 'li ping'
15
```

既然反斜杠字符的特殊作用是表示“转义字符”，那么如何表示反斜杠字符呢？办法是在它前面同样加上反斜杠字符，即转义字符\\表示的是单个斜杠字符\。例如：

```
print('hello \\')
```

输出：

```
hello \
```

转义字符通常可以表示各种不可见的字符，如控制字符。例如，\t 表示“制表符”，\n 表示“换行符”。因为这些字符是不可见的，所以函数 print()在遇到这些字符时会执行特定的动作。例如，对于制表符\t，就输出固定个数的空格，对于换行符\n 就换到新的一行。例如：

```
print('hello\tworld\n')
print('ok')
```

输出：

```
hello world

ok
```

上例在输出的 hello 和 world 之间输出了固定个数的空格，并在换行符处换了一行。

同样可以用**下标运算符[]**访问字符串的一个或多个连续字符。例如：

```
s = 'python programming'
print(s[3])
print(s[1:9])
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
h
ython pr
```

2.3.5 集合

set 是不包含重复元素的无序集合。set 是用左右花括号{}，包围的，以逗号隔开的一组元素。因为集合无序，所以不能用下标操作其中的元素。例如：

```
s = {2,3,'python',8}
print(type(s))
print(s)
```

输出：

```
<class 'set'>
{8, 2, 3, 'python'}
```

set 中不能有相同值的元素。例如：

```
s = {1,2,2,3,3,3}
print(s)
```

输出：

```
{1, 2, 3}
```

集合是根据其元素的哈希值存储元素的，所以无法计算哈希值的对象不能作为集合的元素。例如，list 对象是无法计算哈希值的，所以不能作为集合的元素。例如：

```
s = {2,3,[5,8]}    #list 是一个无法哈希的数据类型
```

产生“list 不是可哈希的类型”的 TypeError(语法错误)：

```
----------------------
TypeError               Traceback (most recent call last)

<ipython-input-28-54a5e1329f94> in <module>()
----> 1 s = {2,3,[5,8]}    #list 不是可哈希的数据类型
TypeError: unhashable type: 'list'
```

2.3.6 字典

dict 是一个“键-值”对(key-value pairs)的无序集合。

dict 中的每个元素都以“键:值(key:value)”的形式存储。例如：

```
d = {1:'value','key':2,'hello':[4,7]}
print(type(d))
print(d)
```

输出：

```
<class 'dict'>
{1: 'value', 'key': 2, 'hello': [4, 7]}
```

需要通过 key(**键**，也称**关键字**)才能访问 dict 中 key 对应的值(value)。例如：

```
d['hello']
```

输出：

```
[4, 7]
```

再如：

```
d[1]
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
'value'
```

总结

- Python 的内在数据类型包括 int、float、complex、bool，str，list、tuple、set、dict 等。
- list 对象是可以修改的，而 str 和 tuple 等对象是不可修改的。
- 对于有序序列数据类型，如 str、list、tuple，可以通过下标访问其中的一个或多个元素，而无序的 set 则不能用下标访问其中的元素。

2.4 类型转换和输入

2.4.1 隐式类型转换

某些情况下，Python 会自动将一个类型转为另外一个类型。例如，在进行整数和浮点数运算时，Python 会自动将整数转为浮点数。同时，两个整数相除时也会自动转为浮点数的相除。例如：


```
a = 25
b = 3.14
c = a+b
print("datatype of a:",type(a))
print("datatype of b:",type(b))
print("Value of c:",c)
print("datatype of c:",type(c))
print(a/3)
```

输出：

```
datatype of a: <class 'int'>
datatype of b: <class 'float'>
Value of c: 28.14
datatype of c: <class 'float'>
8.333333333333334
```

但是，如果一个数(整数、浮点数等)和一个字符串想加，就不会进行“隐式类型转换”。例如：

```
print(30+"world")
```

将产生“不支持 int 和 str 的加法运算”的 TypeError(类型错误)：

```
---------------------------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-2-9869770bfa07> in <module>()
----> 1 print(30+"world")

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

这时需要使用 Python 的内置函数进行“显式类型转换”。

2.4.2 显式类型转换

内置函数 str()将数值类型(int、float、complex)的值转为字符串 str 类型。例如：

```
print(3*str(30)+"world")
print(3*str(3.14))
print( 3*str(3+6j) )
```

输出：

```
303030world
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
3.143.143.14
(3+6j)(3+6j)(3+6j)
```

内置函数 int()既可将一个合适格式的字符串类型的值转为 int 类型的值，也可以将一个 float 类型的值转为 int 类型的值(此时，该值小数点后的部分会被截断)。例如：

```
a = int(3.14)
print(type(a))
print(a)
a = int('1000')
print(type(a))
print(a)
```

输出：

```
<class 'int'>
3
<class 'int'>
1000
```

但是，内置函数 int()不能将一个不合适格式的字符串转为 int 类型的值。

```
a = int('1,000')
```

产生“无效的 int 文字量”的 ValueError(值错误)：

```
---------------------------------------------------------------------------
ValueError                Traceback (most recent call last)
<ipython-input-3-4c775dde81c5> in <module>()
----> 1 a = int('1,000')

ValueError: invalid literal for int() with base 10: '1,000'
```

内置函数 float()既可将一个合适格式的字符串类型的值转为 float 类型的值，也可将一个 int 类型的值转为 float 类型的值。例如：

```
print(30+float('3.14'))
print(float(314))
```

输出：

```
33.14
314.0
```

2.4.3 输入

可以通过内置函数 input()从键盘输入数据，其语法格式为：

```
input(prompt='')
```

其中，prompt 是一个用于提示信息的字符串。例如：

```
name = input("请输入你的用户名：")
print(name)
```

执行上述代码，输出结果如下：

```
请输入你的用户名：小白
小白
```

再如：

```
number = input("请输入一个数：")
print(type(number))
print(number+30)
```

第一个函数 print()打印 number 的类型，第二个函数 print()因 int 和 str 相加而产生 TypeError(语

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

法错误)。输出结果如下：

```
请输入一个数: 34
<class 'str'>
---------------------------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-6-9fef67d3d02c> in <module>()
     1 number = input("请输入一个数: ")
     2 print(type(number))
----> 3 print(number+30)

TypeError: must be str, not int
```

在执行 number+30 时出现了类型错误。函数 input()输入的永远是一个字符串，所以让输入的字符串 34 和 int 类型的值 30 相加是非法的。正确的方法是将输入的代表数值的字符串用 int()或 float()函数转化为数值类型，再和 int 类型的值相加。

```
number = input("请输入一个数: ")
print(float(number)+30)
```

执行：

```
请输入一个数: 34
64.0
```

总结

- 隐式类型转换：表达式中混合 int、float 类型的值时，Python 会自动将 int 类型的值转为 float 类型的值。
- 显式类型转换：可以用内置函数 str()将其他类型的值转为字符串 str 类型，也可以用内置函数 int()或 float()将其他类型的值转为 int 或 float 类型。
- 内置函数 input()从键盘输入的永远是一个字符串 str 类型的值，需要用内置函数 int()或 float()转为数值类型才能参与算术计算。

2.5 运算符和表达式

2.5.1 运算符和表达式的应用方法

程序就是对数据进行处理，表示数据的值(对象)可以用变量给它们起名字(用变量名引用这些值(对象))，对这些变量或值进行处理可使用一些特殊的符号如+、*等进行加法、乘法等运算，这些表示不同运算功能的特殊符号称为**运算符**(operators)。这些运算符操作的变量或值称为“**运算数**(operands)”。例如：

```
2+3
3/(5+2)
3*"world"
```

用+、/、*、(、)这些运算符对运算数进行运算的式子就是**表达式**(expression)，即表达式是由运算数(值、变量、对象)和运算符构造的。

因为这些表示运算的运算符来自人们熟悉的符号，用它们构成的表达式的含义都很清楚。

前面介绍过，不同类型的对象支持的运算符是不同的，如不能对 2 个 str 字符串类型的变量应用–、*、/、//等运算，也不能对 1 个字符串和 1 个整数进行如–、+、/等运算。例如，下面的都是非法的：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
'hi'-1        'Hello'/123          'hi'*'Hello'        '15'+2
```

即使可以对 2 个字符串应用+运算符，其含义也不同于 2 个整数的加法运算，而实际是对 2 个字符串进行拼接。同样，对 1 个字符串和 1 个整数可以应用乘法运算符*，其含义是产生一个重复复制的字符串。例如：

```
s1 = "hello"
s2 = s1+"world"
s3  =3*"world"
print(s2)
print(s3)
```

输出：

```
helloworld
worldworldworld
```

不同运算符具有不同的优先级和结合性。例如：

```
3+5/2     #优先级：先乘除后加减
3+5-2     #结合性：从左向右计算
```

“3+5/2”中运算符/优先于+，“3+5–2”中+和–具有同样的优先级，但它们的结合性都是“从左向右”，因此，先计算左边的+，再计算右边的–。

2.5.2 运算符的种类

和其他编程语言一样，Python 有不同功能的各种运算符，如进行数学计算的“**算术运算符**”，比较 2 个运算数大小的“**比较运算符**”，进行逻辑运算的“**逻辑运算符**”，对二进制位进行操作的“**二进制运算符**”，有对变量赋值的“**赋值运算符**”，还有一些特殊的运算符。

1．算术运算符(arithmetic operators)

表 2-1 是算术运算符的含义及其示例。需要注意的是，//表示整数除法，而/表示的是浮点除法，%用于求 2 个整数相除的余数。例如：

```
x = 15
y = 2
print('x + y =',x+y)
print('x - y =',x-y)
print('x * y =',x*y)
print('x / y =',x/y)
print('x // y =',x//y)
print('x ** y =',x**y)
print('x % y =',x%y)
```

表 2-1 算术运算符的含义及其示例

运算符	含　义	示　例	结　果
+	加	15+2	17
–	减	2–15	–13
*	乘	15*2	30
/	浮点除	15/2	7.5
//	整数除	15//2	7
**	幂	15**2	225
%	求余	15%2	1

输出：

```
x + y = 17
x - y = 13
x * y = 30
x / y = 7.5
x // y = 7
x ** y = 225
x % y = 1
```

再如：

```
print(15.3/2.5)
print(15.3//2.5)      #转为整数除，相当于 15//2
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print(15.3%2.5)       #15.3 除 2.5 的余数
print(15.3**2.5)      #15.3 的 2.5 次方
```

输出：

```
6.12
6.0
0.3000000000000007
915.6480546203329
```

2．比较运算符(comparison operators)

表 2-2 是比较运算符及其含义和示例。

对两个量进行比较，产生的结果是一个表示“真”(True)或“假”(False)的 bool(布尔)类型的值。例如：

```
x = 15
y = 2
print('x > y  is',x>y)
print('x < y  is',x<y)
print('x == y is',x==y)
print('x != y is',x!=y)
print('x >= y is',x>=y)
print('x <= y is',x<=y)
```

输出：

```
x > y  is True
x < y  is False
x == y is False
x != y is True
x >= y is True
x <= y is False
```

表 2-2　比较运算符的含义及其示例

运算符	含　义	示　例	结　果
>	大于	15>2	True
<	小于	15<2	False
==	等于	15==2	False
!=	不等于	15!=2	True
>=	大于等于	15>=2	True
<=	小于等于	15<=2	False

3．逻辑运算符(logical operators)

逻辑运算符 and、or、not 分别表示逻辑与、逻辑或、逻辑非运算。

在逻辑运算中，True、非 0 或非空对象就是真(True)，而 False、0 或空对象就是假(False)。

当一个对象 x 是真(True、非 0 值或非空值)时，not x 就是 False，当 x 是假(False、0 或空值)时，not x 就是 True(真)。例如：

```
print(not 0)
print(not [])
print(not False)
print(not 2)
print(not [3])
print(not True)
```

输出：

```
True
True
True
False
False
False
```

假设有 2 个对象 x、y，当 x 是真时，x or y 的结果就是 x，当 x 是假时，x or y 的结果就是 y。例如：

```
print(3 or 5)      #因为 3 是真，所以 3 or 5 的结果就是 3
print(3 or 2)      #因为 3 是真，所以 3 or 2 的结果就是 3
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print(0 or 2)      #因为 0 是假，所以 0 or 2 的结果就是 2
print(False or True)
print(False or [])
print({} or 2)     #因为{}是假，{} or 2 的结果就是 2
```

输出：

```
3
3
2
True
[]
2
```

假设有 2 个对象 x、y，当 x 是真时，x and y 的结果就是 y，当 x 是假时，x and y 的结果就是 x。例如：

```
print(3 and 5)      #因为 3 是真，所以 3 and 5 的结果就是 5
print(3 and 2)      #因为 3 是真，所以 3 and 2 的结果就是 2
print(0 and 2)      #因为 0 是假，所以 0 and 2 的结果就是 0
print(False and True)
print(False and [])
print({} and 2)     #因为{}是假，所以{} and 2 的结果就是{}
```

输出：

```
5
2
0
False
False
{}
```

逻辑运算符 and、or、 not 的含义及其示例如表 2-3 所示。

表 2-3　逻辑运算符的含义及其示例

运　算　符	含　义	示　例	结　果
and	与	x and y	当 x 是真时，x and y 的结果就是 y；当 x 是假时，x and y 的结果就是 x
or	或	x or y	当 x 是真时，x or y 的结果就是 x；当 x 是假时，x or y 的结果就是 y
not	非	not x	当 x 是真时，not x 的结果就是 False；当 x 是假时，not x 的结果就是 True

这 3 个逻辑运算符具有不同的优先级，and 优先于 or，not 优先于 and、or。例如：

```
print(2 or 5 and 3)
print( (2 or 5) and 3)  #可用圆括号()改变运算符的计算次序
```

输出：

```
2
3
```

4．位运算符(bit operators)

任何量在计算机内存里都是用 1 串二进制位表示的。

假如有 2 个数，a = 37 和 b = 22 是用 1 字节表示的，则其内存的二进制位如下所示：

```
a = 0 0 1 0 0 1 0 1
b = 0 0 0 1 0 1 1 0
```

“位运算”：对运算数的二进制位进行相应的运算。其中，&(位与)、|(位或)、^(异或)是对 2 个运算数的对应二进制位进行运算，而~(取反)、<<(左移位)、>>(右移位)则是对 1 个运算数的二进制位进行运算。

二元位运算符的运算规则如表 2-4 所示。3 个二元位运算符的运算规则如下。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- &(位与)运算：只有 p 和 q 都是 1 时，p&q 的结果才是 1；如果有一个为 0，则结果是 0。
- |(位或)运算：只要 p 和 q 有一个是 1 时，p|q 的结果就是 1；如果 p 和 q 都为 0，则结果是 0。
- ^(异或)运算：当 p 和 q 不同(一个是 0，另一个是 1)时，p^q 的结果是 1，否则结果是 0。

表 2-4　二元位运算符的运算规则

p	q	p&q	p|q	p^q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

例如，对 a、b 的&、|、^运算是对它们的对应二进制位进行运算：

```
a&b  = 0 0 0 0 0 1 0 0
a |b = 0 0 1 1 0 1 1 1
a^b  = 0 0 1 1 0 0 1 1
```

下面是对 a、b 的&、|、^运算的代码。

```
a = 37
b = 22
print('a = ','{:08b}'.format(a))
print('b = ','{:08b}'.format(b))
print('a&b=','{:08b}'.format(a&b))
print('a|b=','{:08b}'.format(a|b))
print('a^b=','{:08b}'.format(a^b))
```


str 的 format()函数格式化输出 a、b、a&b、a|b、a^b 的二进制字符串。输出如下：

```
a =  00100101
b =  00010110
a&b= 00000100
a|b= 00110111
a^b= 00110011
```

一元位运算符取反~、左移<<、右移>>的运算规则如下。

- ～(取反)：将每个二进制取反(0 变成 1，1 变成 0)。例如，对 x，~x 的结果相当于–x–1，如 22 的补是–23。
- << (左移)：各二进制位全部左移若干位，高位丢弃，低位补 0。
- >> (右移)：各二进制位全部左移若干位，无符号数，高位补 0。

例如，对 b 的<<、～运算的结果如下：

```
b<<2= 0 1 0 1 1 0 0 0
~b  = 1 1 1 0 1 0 0 1
```

下面代码输出了对 b 的<<、>>运算的二进制表示：

```
print('b =   ','{:08b}'.format(b))
print('b<<2: ','{:08b}'.format(b<<2))      #b 的二进制左移 2 位，右边低位补充 0
print('b>>2: ','{:08b}'.format(b>>2))      #b 的二进制右移 2 位，左边高位补充 0
print('~b:   ','{:08b}'.format(~b))        #~b 就是-(b+1)，b=22 的补是-23
print(b)
print(b<<2)
print(b>>2)
print(~b)
```

输出如下：

```
b =   00010110
b<<2:  01011000
b>>2:  00000101
~b:   -0010111
22
88
5
-23
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

其中的取反运算～涉及 1 个数在计算机内部的二进制表示。数字在计算机中是以补码保存的，所以 Python 位运算是作用在补码上的。假如，用 8 位二进制表示，正整数 1，其二进制的原码是 00000001，而负整数–1，其二进制的原码是 10000001，即左边的最高位 1 表示这是一个负数。除原码外，1 个数在计算机内部还可以用反码和补码表示。正数的反码与其补码和原码是一样的，如 1 的反码和补码都是 0000001，而负数的反码除左边最高位外，其他位都是原码的取反，如–1 的反码是 11111110，而补码就是在反码基础上加上 1，即–1 的补码是 11111111。

b=22 的原码、反码和补码都是 00010110。对 b 的二进制取反的结果就是 11101001，这个数实际上是–(b+1)的补码。因为–b 的二进制原码是 10010110，而–(b+1)的二进制原码是 10010111，其补码就是 11101001。但 bin()和 str 的 fromat()会将～b 的补码表示转换成原码表示，因此，尽管～b 运算产生的二进制是 11101001，但显示的仍是其二进制原码 10010111。

5．赋值运算符

赋值运算符=是给 1 个对象起 1 个名字，如简单的赋值运算符=的语句“x=3”，就是给对象 3 起了一个变量名 x。例如：

```
x = 2
x = x+3
```

在表达式“x = x+3”中，右边的变量 x 引用的是原先的对象，而左边的变量 x 引用的是新的结果对象。

简单赋值运算符可以和算术、位运算结合构成复合赋值运算符。例如：

```
x=2
x+=3
```

“x += 3”实际上是“x = x+3”的简写，即“将 x+3 的结果赋值给 x”，也就是说，给 x+3 的结果对象起了一个名字 x。变量 x 现在引用的是一个新的结果对象而不是原来的 2，可以通过输出 x 的 id 来验证这一点：

```
x=2
y = x      # y 和 x 都是对象 2 的名字
print(x)
print(id(x))
x**=3   #相当于 x = x**3
print(x)
print(id(x))
print(y)
print(id(y))
```

输出：

```
2
1477476432
8
1477476624
2
1477476432
```

Python 的赋值运算符如表 2-5 所示。

6．成员运算符(in、not in)

成员运算符 in 和 not in 用于判断一个值(对象)是否在一个容器对象，如 list、str、tuple 对象中，这 2 个运算符返回的结果是 True 或 False。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
alist = [2,5,7,8]
print(3 in alist)
print(5 in alist)
print(3 not in alist)
print(5 not in alist)
```

输出：

```
False
True
True
False
```

7．身份运算符：is、is not

身份运算符 **is** 和 **is not** 用于判断 2 个变量(标识符)是不是引用的同一个对象。例如：

```
a = 10
b = 10
print(id(a))
print(id(b))
print(a is b)
print(a is not b)
```

输出：

```
1477476688
1477476688
True
False
```

再如：

```
a = 1000
b = 1000
print(id(a))
print(id(b))
print(a is b)
print(a is not b)
```

输出：

```
2417203138160
2417203138192
False
True
```

这是因为，Python 解释器执行时会将小整数驻留在内存中，将小整数赋值给多个变量，它们实际引用的是同一个对象。而将整数赋值给不同变量，这些变量将引用临时创建的不同对象。

表 2-5　赋值运算符

运　算　符	示　　例	等　价　于
=	x = 3	x = 3
+=	x += 3	x = x + 3
–=	x –= 3	x = x – 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

2.5.3　运算符的优先级

1．运算符的优先级

当一个表达式中有多个运算符时，这些运算符的计算是有先后次序的，即运算符是具有不同的**优先级**的。例如，算术运算符总是“先乘除，后加减”，同样级别的运算符，如+和–则按照“自左往后的次序”计算。例如：

```
3+5/2-4
```

先做除法“5/2”，然后做加法，即用“3”和“5/2”的结果相加，最后才做减法，即前面加法的结果和 4 相减。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

再比如，同样是逻辑运算，not 比 and 优先级高，而 and 比 or 优先级高。

实际上，无须记忆这些运算符的优先级，如果不清楚不同运算符的优先级，则可以简单地用圆括号，使其按照希望的意图进行计算。例如：

```
(3+5)/(2-4)
```

2．运算符的优先级表

表 2-6 是 Python 运算符的优先级表(从上到下优先级从高变低)。

表 2-6　Python 运算符的优先级表

运　算　符	描　　述
()	圆括号
f(args)	函数调用
x[index:index]	切片
x[index]	下标
x.attribute	属性引用
**	指数
~x	补
+x、-x	一元正号、一元负号
*、/、%	乘、除、求余
+、-	加、减
<<、>>	位左移、位右移
&	位与
^	异或
|	位或
in、not in、is、not is、<<=、 >、 >=、<>、!===	比较、成员、身份
not x	逻辑非
and	逻辑与
or	逻辑或
lambda	lambda 表达式

总结

- 表达式是用运算符对运算数进行运算的式子，运算数包括值或变量，变量实际就是值对象的引用。
- 常见的运算符按照其功能，主要包括算术运算符、逻辑运算符、比较运算符、位运算符。还有一些其他的运算符，如比较运算符(in 和 not in)、身份运算符(is 和 is not)。
- 不同的运算符具有不同的优先级，优先级高的先运算。无须记忆运算符的优先级，因为可以用圆括号保证正确的运算顺序。

2.6　可变对象和不可变对象

Python 的数据类型分为可变类型和不可变类型，可变类型的对象的值是可以被修改的，称为**可变对象**，而不可变类型的对象的值是不可修改的，称为**不可变对象**。数值类型、str 类型、tuple 类型都是不可变类型，而 list 类型、set 类型、dict 类型等都是可变类型。例如，一个 list 对象的值是可以修改的，即可以对其中的元素进行修改、删除、添加等操作。例如：

```
a = [1,2,3]
a[0] = 4
print(a)
```

输出：

```
[4, 2, 3]
```

可以看到，变量 a 引用的 list 对象的第 1 个元素被修改了。因为 tuple 对象是不可修改的，下面代码将产生错误：

```
t = (1,2,3)
t[0] = 4
```

产生了类型错误“TypeError: 'tuple' object does not support item assignment”。

但下面代码不会有任何错误：

```
t= (1,2,3)
print(t)
t =(4,5,6)
print(t)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
(1, 2, 3)
(4, 5, 6)
```

上述代码中的变量名分别指向了不同的 tuple 对象，而它们指向的 tuple 对象本身并没有被修改。

前面讲过，变量是对象的引用，也就是说，变量仅是对象的别名，而不是对象本身。当变量 t 引用不可修改的 tuple 对象(1,2,3)时，这个 tuple 对象不可修改，但可以对变量名 t 重新赋值，如“t=(4,5,6)”使变量 t 又引用了这个新的 tuple 对象。

可以通过下面代码进一步理解上述内容：

```
x = "hello"
y = x
print(x is y)          #x,y 指向同一个不可修改对象 hello
print(id(x),id(y))
x = "world"            #x 引用了新对象 world，而并未修改对象 hello
print(y)               #y 仍然引用的是对象 hello
print(x)               #x 引用的是新对象 world
print(id(x),id(y))
```

输出：


```
True
1793971891984 1793971891984
hello
world
1793971891480 1793971891984
```

可以看到，变量 x 前后指向的是不同的对象，而并未修改变量 x 指向的对象。

同样，执行复合赋值语句，如“x+=1”等，并不修改 x 指向的对象，而是让 x 指向新创建的对象。例如：

```
x = 2
y = x          #y 和 x 指向了同 1 个对象 2
print(id(x),"\t", id(y))
x+=1           #x+1 等价于 x = x+1。因为 x 指向的 int 对象是不可修改的(immutable)
               #用 x+1 的结果给 x 赋值，就是让 x 变量名指向这个新的结果值对象
print(x)
print(y)
print(id(x),"\t",id(y))
```

其中，“x+=1”使得 x 引用了新对象，而并没有修改原来 x 引用的不可变对象“2”。

输出：

```
1717202432    1717202432
3
2
1717202464    1717202432
```

如果 1 个变量指向的是 1 个可修改的(mutable)对象，那么可以通过这个变量修改它指向的对象，而这个时候并不会创建新对象。例如：

```
a= [2,3]
b = a          #b 和 a 都是 list 对象[2,3]的引用
a[0] = 4       #修改 a 指向的 list 对象的第 1 个元素
print(a)
print(b)
print(id(a))
print(id(b))
b[1] = 7
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print(a)
print(b)
```

输出：

```
[4, 3]
[4, 3]
2214058333320
2214058333320
[4, 7]
[4, 7]
```

可以看出，上述代码修改 a、b 指向的 list 对象的元素，并没有修改 a、b 本身。因此，a、b 始终指向的是同 1 个对象，无论变量指向的是 1 个可修改的对象还是不可修改的对象，给这个变量赋值都会使这个变量引用其他的对象。例如：

```
a = [2,3]
print(id(a))
a = [4,5]
print(id(a))
```

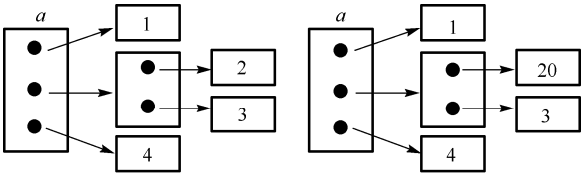
输出：

```
1810380911112
1810363678280
```

再看下面的代码：

```
a= (1, [2,3], 4)
b = a
#a[1] = 20        #错：元祖 a 的元素是不可修改的
print(a[1])
print(id(a[1]))
a[1][0]=20        #a[1]元素是一个 list 对象，是可修改的
print(a[1])
print(id(a[1]))
```

因为元组 a 是不可修改的，即不可以修改其内容如元素 a[1]的值，即“a[1]=20”将出错，但可以修改 a[1]引用的那个可变 list 对象[2,3]。因此，“a[1][0]=20”是可以修改的。通过输出 a[1]的 id 值，可以发现，作为元素 a 的元素，a[1]的值没有改变，仍然引用的是原先的 list 对象，但其引用的 list 对象内容发生了变化(通过打印 a[1]可知道这一点)。以上修改 a[1]引用的 list 对象的。过程如图 2-2 所示。



(a) 修改前　　(b) 修改后

图 2-2　修改 a[1]引用的 list 对象

总结

- 数据类型分为可变类型和不可变类型，可变类型的对象的值是可修改的，而不可变类型的对象的值是不可修改的。
- 无论变量指向的是一个可修改的对象，还是一个不可修改的对象，给这个变量赋值都会使这个变量引用其他的对象(如创建的新对象)。对变量指向的可变对象的值进行修改不会使变量引用新对象，只是修改变量引用的对象的值。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2.7 控制语句

2.7.1 if 条件语句

1. if 条件语句

前面讲解的程序语句都是按照书写的顺序自上而下一条一条地执行，但有时需要根据不同情况选择执行不同的语句。例如，输入学生成绩，根据成绩是否小于 60 打印“及格”或“不及格”，在 Python 程序中，可以用 if 语句实现。例如：

```
score= float(input("请输入学生成绩："));
if score>=60:
    print("恭喜你!")
    print("通过了考试。")
```

输出：

```
请输入学生成绩：60.5
恭喜你!
通过了考试。
```


上述程序使用了最简单的 **if 语句**(也称 **if 条件语句**，英文 if 的意思是“如果”)。其语言格式如下：

```
if 表达式:
   程序块代码
```

if 关键字后面跟 1 个表达式和冒号:，然后是缩进的语句(程序块)。其含义是，如果“表达式”的值是 True，则执行 if 语句体，即缩进的程序块代码；如果“表达式”的值是 False，则不会执行缩进的程序块代码。

2. 代码缩进

if 语句体的多条语句必须具有一致的缩进，即这些语句的第 1 个字母必须对齐，且不能和 if 关键字对齐。例如，下面的 if 语句没有缩进因此语法是错误的：

```
if score>=60:
print("恭喜你!")
print("通过了考试。")
```

执行上述语句会产生“缩进错误(IndentationError)”：

```
  File "<ipython-input-2-362eeb5f9c61>", line 2
    print("恭喜你!")

IndentationError: expected an indented block
```

同样，if 语句体中语句不具有“一致的”缩进也是错误的。例如：

```
if score>=60:
print("恭喜你!")
 print("通过了考试。")
```

if 语句还有另外一种形式(**if**…**else**)：

```
if 表达式:
   程序块 1
else:
   程序块 2
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

上述形式中，“表达式”的值如果为 True，则执行“程序块 1”中的代码，如果为 False，则执行“程序块 2”中的代码。英文 **else** 的意思是“否则”。

例如：

```
score= int(input("请输入学生成绩："));
if score>=60:
    print("恭喜你!")
    print("通过了考试。")
else:
    print("很可惜！")
    print("你未通过考试。")
    print("继续努力，加油！")
```

这段代码的意思是，如果达到了 60 分就显示祝贺的信息，否则，就鼓励学生继续努力。

if 语句还有另外一种形式(if…elif…else)：

```
if 表达式 1:
   程序块 1
elif 表达式 2:
   程序块 2
elif 表达式 3:
   程序块 2
else:
   程序块 k
```

对多个不同的条件依次进行判断时，首先判断 if 里的“表达式 1”是否为 True，如果为 True 就执行“程序块 1”，否则就继续判断下面的 elif 的“表达式 2”是否为 True，如果为 True 就执行“程序块 2”，否则就继续判断下面的 elif 的“表达式 3”是否为 True，…，依次判断下去，如果前面的表达式都不是 True，就执行最后的 else 里的“程序块 k”。

英文 elif 的意思是“否则如果”。例如：

```
score= int(input("请输入学生成绩："));
if score<60:
    print("不及格")
elif score<70:
    print("及格")
elif score<80:
    print("中等")
elif score<90:
    print("良好")
else:
    print("优秀");
```

这段代码根据输入的不同的百分制分数，输出 5 个等级中的 1 个。

if ...else…还有如下的一种用法：

```
A if B else C
```

其含义是，如果 B 成立就执行 A，否则就执行 C。例如：

```
a,b = int(input("a")),int(input("b"))
print("a>b") if a>b else  print("!(a>b)")
```

执行上述代码，输入 3 个整数的运行情况：

```
a3
b5
!(a>b)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2.7.2 循环语句

有时需要在某种条件满足的情况下重复执行某个语句(块)。例如，要编写一个程序统计学生的平均成绩，因为不知道程序使用时实际的学生人数，通常会让用户从键盘或文件不断输入学生成绩数据，并且将输入的分数累加为总分数，最后在某种条件下结束这个输入过程，并将总分数除总人数，从而得到平均成绩。

再例如，要计算一组数的和，如 1+2+3+4 的和，可以简单地写出所有数，通过+将它们累加起来：

```
print(1+2+3+4)
```

但如果要计算 1 到 1000 之间的所有整数之和，那么写出所有这些数字的连加是不现实的，当然也不能写成“1+2+3+…+1000”形式，那么如果计算呢?

这就需要用到**循环语句**(也称**迭代语句**)：当某个条件成立时，就一直重复执行某块程序代码(循环体中的代码)。

1．while 循环语句

while 循环语句的语法格式如下：

```
while 表达式:
    程序块
```


当关键字 while 后的“表达式”为 True，就重复执行循环体中的“程序块”。例如：

```
i = 1
s = 0
while i<=1000:
   s = s+i;      #等价于  s += i
   i+=1
print(s)
```

只要 i 不超过 1000，就一直执行由“s = s+i”和“i+1”构成的 while 循环语句的循环体。当 i>1000 时，while 语句才真正执行完。

例如，可以用 while 循环语句编写计算学生平均成绩：

```
total_score=0
i= 0
score = float(input("请输入学生成绩："))
while score>=0:
    total_score += score
    i += 1
    score = float(input("请输入学生成绩："))
print('平均成绩为：', total_score/i)
```

该程序只要输入的学生分数大于等于 0，就一直执行循环体，当输入分数为负数时，就停止循环，输出最后的平均成绩“total_score/i”。

2．for 循环语句

Python 还有一个用于遍历访问一个容器(如 str、list 或 tuple 等容器)对象的每个元素的 for... in 循环。其语法格式是：

```
for  e  in 可迭代对象
       对 e 的处理语句块
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

其含义是依次访问可迭代对象(iterable)中的每个元素 e，并对该元素 e 用 for 循环体的语句块进行处理。例如：

```
numbers = [23, 41,19,87,67,2,32]
sum = 0
# 对列表中的每个元素 val
for val in numbers:
   sum = sum+val
print("The sum is", sum)   # 输出和 sum
```

该循环将 list 对象 numbers 的每个整数累加到 sum 中，最后输出这个 sum：

```
The sum is 271
```

再如：

```
words = ['cat', 'dog', 'book']
for w in words:
   print(w, len(w))
for c in "world":
   print(c)
```

输出：

```
cat 3
dog 3
book 4
w
o
r
l
d
```

注：可迭代对象(将在本书第 8 章介绍)是可以通过 for 循环迭代地访问其中的元素的一种对象，如前面的容器 list、tuple、set、dict 和字符串 str 类型的对象都是可迭代对象。上面的例子中通过 for 可访问一个 str 对象中的每个字符。例如，内置函数 range()可以产生 1 对整数之间的 1 个可迭代对象：

```
for e in range(2,8):
   print(e,end=" ")
```

输出：

```
2 3 4 5 6 7
```

函数 range()的语法格式是：

```
range(start, stop[, step])
```

该函数返回一个可迭代对象，该对象包含了介于 start 和 stop 之间(但不包含 stop)的一系列整数的可迭代对象：

```
start、start+step、start+step+step、…
```

可以将函数 range()产生的可迭代对象传递给内置函数 list()或 tuple()等构造一个 list 对象或 tuple()对象。例如：

```
L = list(range(2,8,2))
print(L)
t = tuple(range(2,-8,-2))
print(t)
```

输出：

```
[2, 4, 6,]
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
(2, 0, -2, -4, -6)
```

3．break 语句

在重复执行循环体时，如果满足某种条件，就可以用 break 语句跳出循环，即不再执行循环体。

例如，前面的计算 1 到 1000 之间的整数和、统计学生平均成绩等程序都可以改写为用 break 语句终止循环体的执行。例如：

```
i = 1
s = 0
while True:       #条件永远满足
   s = s+i;       #等价于  s+=i
   i+=1
   if i>1000:     #i 大于 1000 时，跳出循环体
      break;
print(s)
```

该代码当 i>1000 时，用 break 语句跳出 while 语句，然后执行 print()。

输出：

```
500500
```


4．continue 语句

Python 还提供了一个在循环体中使用的 **continue** 语句。在执行循环体时，如果遇到 **continue** 语句，则会停止执行其后续语句，返回循环开头继续执行循环体。

例如，以下程序计算一个整数列表中的所有非负整数之和：

```
#计算 list 中的所有非负数之和
numbers = [23, 41,-19,87,-3,67,2,-32]
sum = 0

for val in numbers:
    if val<0:
        continue
    sum += val
print("The sum is", sum)   # 输出和 sum
```

当循环体中遇到 val 小于 0 的数，就用 continue 语句返回循环开始，而不会执行其后面的“sum += val”，也就是说，负数就不累加，只累加非负数。

5．else 语句

循环语句(for 或 while)都可以在后面跟一个 else 语句，当 for 语句遍历完整个可迭代对象(如遍历完整个 list 的元素)，或者当 while 语句的条件表达式是 false 退出时，都会执行这个 else 语句。但是，如果是通过 break 跳出循环语句则不会执行这个 else 语句。例如：

```
alist = [3, 1, 5]
for i in alist:
    print(i)
else:
    print("已经访问了列表 alist 的所有元素。")
```

输出：

```
3
1
5
已经访问了列表 alist 的所有元素
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2.7.3 pass 语句

pass 是一个空语句，Python 程序有的地方需要一条语句，但是又不希望这条语句执行任何动作，这时就可以用 pass 语句，使程序不会产生语法错误。或者，编写程序时还未想好写什么代码，此时可以临时用一个 pass 语句作为“占位符”，等以后再补充添加代码语句。例如：

```
score = float(input("输入一个成绩："))
if score <0:
   pass   #没有这个 pass，程序就会出错
else:
   print("分数是：", score)
```

总结

- if 条件语句根据条件表达式是否满足而执行不同的程序块，有 if、if...else、if...elif...else 等多个变化用法。
- while 循环语句是当条件表达式为 True 时，会一直执行循环体中的程序块。循环 for 语句是迭代访问一个**可迭代对象**的所有元素。
- break 语句用于跳出循环。continue 语句用于终止循环体中后续的语句执行，重新开始下一轮循环。除非用 break 语句跳出循环，否则循环结束总会执行 else 语句。
- 空语句 pass 用于语法上需要语句而实际不需要做任何工作的地方，或者用作占位符，以等待后续补充代码。

2.8 实战

2.8.1 二分查找

1. 顺序查找

若要查找一个序列中是否存在某个元素，一般可以采用遍历的方法查找。例如：

```
arr = [12,46,25,43,58,7,37,5,80,34,105,82]
v = int(input("输入你要查找的数"))
ok = False
for e in arr:
   if e==v:
      ok = True
      break
if ok:
   print('查找成功！')
else:
   print('查找失败！')
```

执行：

```
输入你要查找的数 6
查找成功!
```

分析这个程序的时间效率。如果一个序列中数据元素的个数为 n，且每个元素被查找的概率是均等的(1/n)，在查找成功的情况下，第 1、2、3、…、n 元素的比较次数分别为 1、2、3、…、n，因此，查找成功的情况下查找一个元素的平均比较次数是 1(1/n)+2(1/n)+...+n(1/n) = (n+1)/2，即平均需要比较的次数是表长的一半。当 n 趋向于无穷大时，(n+1)/2 和 n 在一个数量级，所以称其**时间复杂度**是 O(n)，即查找时间和 n 是同阶无穷大量。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

假如 n = 1024，则成功查找其中的一个元素需要平均比较 512 次。但如果这个序列是有序的，那么可以使用“**二分查找**”的算法，平均只需要 $\log_2(1024)$ 次，即 10 次就可以找到该元素。

2．二分查找

“二分查找”的思想很简单。例如，在一个有序序列 alist 中查找某个元素 item，则可以让 item 首先和 alist 的中间元素比较：

- 如果相等，则成功；
- 如果小于中间元素，则在 alist 的左半区间查找；
- 如果大于等于中间元素，则在 alist 的右边半区间查找。

如图 2-3 所示，是在一个有序序列中查找 25 的过程。

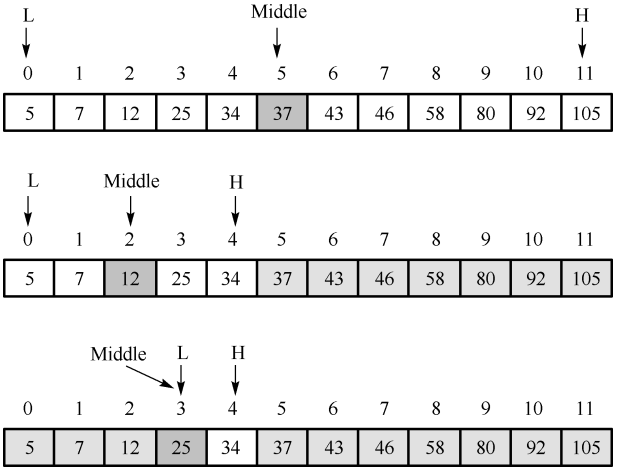


图 2-3　查找 25 的过程

以上查找过程是利用一个循环迭代过程，通过不断更新区间的左右位置 L、H 和中间位置 Middle 这 3 个指示器来查找。程序如下：

```
def binarySearch(alist, value):
    L = 0                           #区间左端点
    H = len(alist)-1                #区间右端点
    found = False
    while L<=H:                     #区间(L，H)存在
        Middle = (L+H)//2           #Middle 指向区间的中点
        if alist[Middle] == value: #等于 Middle 指向的元素，找到了
            return Middle
        else:
            if value < alist[Middle]:
                H = Middle-1        #在左区间查找
            else:
                L = Middle+1        #在右区间查找
    return -1

testlist = [5,7,12,25,34,37,43,46,58,80,82,105]
print(binarySearch(testlist, 25))
print(binarySearch(testlist, 13))
```

输出：

```
3
-1
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2.8.2 冒泡排序和简单选择排序

1. 冒泡排序

冒泡排序的思想是对相邻元素比较大小，如果逆序就交换它们。冒泡排序如图 2-4 所示，对于一个序列，通过这种两两相邻元素的比较与交换，可以将最大值(或最小值)放在最后一个位置，这一过程称为“**一趟冒泡**”。“一趟冒泡”只是在一个序列中选出了一个最大值(或最小值)并放在了其最终位置。对于剩余元素构成的序列，再进行“一趟冒泡”，又可以在剩余元素序列中选出一个最大值(或最小值)并放在剩余元素序列的最终位置(如倒数第二个位置)。重复这个过程，直到剩余一个元素位置。对于 n 个元素的序列，需要进行 n–1 趟冒泡。程序如下：

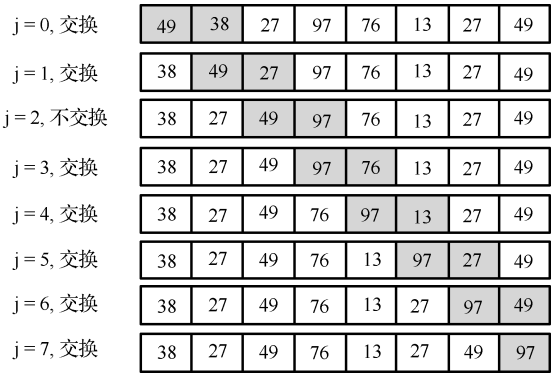


图 2-4 冒泡排序

```
alist = [49,38,27,97,76,13,27,49]
debug = True

for i in range(len(alist)-1,0,-1):
       for j in range(i):
           if alist[j]>alist[j+1]:
               #temp = alist[j]
               #alist[j] = alist[j+1]
               #alist[j+1] = temp
               alist[j],alist[j+1] = alist[j+1],alist[j]  #交换两个元素
       if debug:
          print(alist)
print(alist)
```

输出：

```
[38, 27, 49, 76, 13, 27, 49, 97]
[27, 38, 49, 13, 27, 49, 76, 97]
[27, 38, 13, 27, 49, 49, 76, 97]
[27, 13, 27, 38, 49, 49, 76, 97]
[13, 27, 27, 38, 49, 49, 76, 97]
[13, 27, 27, 38, 49, 49, 76, 97]
[13, 27, 27, 38, 49, 49, 76, 97]
[13, 27, 27, 38, 49, 49, 76, 97]
```

2. 简单选择排序

简单选择排序的思想是：在整个序列中选出一个最值(如最小值)放在序列的开头(或结束)位置。这个过程称为“**一趟选择**”。对于剩余元素构成的序列重复这个过程，又选出一个最值放在剩余元素序列的开头(或结束)位置，即“第二趟选择”。这个过程一直进行下去，直到剩余序列只有一个元素。请读者根据这个思想写出简单选择排序的程序。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2.8.3 Floyd 最短路径算法

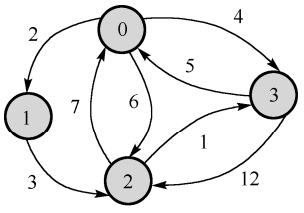
1. 最短(最佳)路径问题

最短路径问题是日常生活和科学研究中广泛应用的问题。例如，一个人在一个陌生城市，要从某一个地点前往另外一个地点，此时他会打开手机中的导航软件，导航软件会按照不同代价(时间、路程、花费)给出不同的最佳(最短)路径。再如，计算机网络通信需要在不同计算机之间发送数据包，网络路由算法会采用最短路径算法计算出最佳的数据包发送路径。

如图 2-5(a)所示的一个城市公路图。其中，顶点表示城市，而边表示城市之间的公路及其距离，现在要求出任何两个城市之间的最短路径。最短路径属于图论的一个基本问题，针对这个问题有很多最短路径算法，其中的 Floyd 算法可以求出任意两个顶点之间的最短路径。

2. Floyd 算法

Floyd 算法的基本思想是：用一个二维矩阵(如图 2-5(b)所示)表示任意两个顶点之间的距离，初始时，这个矩阵的数据元素的值表示的是两个城市的直达距离，值是无穷大∞则表示没有直达距离。



(a) 城市公路图

	0	1	2	3
0	0	2	6	4
1	∞	0	3	∞
2	7	∞	0	1
3	5	∞	12	0

(b) 二维矩阵

图 2-5　城市公路图及其邻接矩阵

直达距离不一定是最短距离(顶点 0 到顶点 2 的直达距离 6 不是顶点 0 到顶点 2 的最短距离)，因此，Floyd 算法每次考虑绕道一个顶点，查看是否有两个顶点的距离会因为绕道这个顶点变得更近。

假如当前的距离矩阵为 D，现在绕道顶点 w，查看顶点 u 到顶点 v 之间的距离 D[u][v]会不会因为绕道顶点 w 变得更短。如果更短，则更新 D[u][v] = D[u][w]+ D[w][v]，即：

```
if   D[u][w]+ D[w][v] < D[u][v]:
     D[u][v] =  D[u][w]+ D[w][v]
```

对这个绕道顶点 w，检查所有其他的顶点 u、v 是否因为绕道 w 而变得更短，从而更新距离矩阵 D。

不断重复绕道不同的顶点，并更新这个距离矩阵 D，直到所有顶点都绕道完为止，得到的最终矩阵就是这两个顶点的最短距离。

为了记录路径，需要用一个和 D 一样大小的二维矩阵 P 记录任意两个顶点之间的当前距离对应的路径上的倒数第二个顶点(路径上终点之前的那个顶点)。例如：

```
if   D[u][w]+ D[w][v] < D[u][v] :
     D[u][v] =  D[u][w]+ D[w][v]
     P[u][v] =  P[w][v]        #P[u][v] =  P[u][w]+ P[w][v]
```

下面是 Floyd 算法的 Python 代码实现：

```
n = 4
INFINITY = 100000.0                    #假设这是一个很大的数值

D=[[0,2,6,4],                          #距离矩阵
   [INFINITY,0,3,INFINITY],
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        [7,INFINITY,0,1],
        [5,INFINITY,12,0]]

P = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]    #路径矩阵
//初始化路径矩阵，即 P[u][v] 记录直达路径 uv 的终点 v 前面的顶点 u
for u in range(0,n):
    for v in range(0,n):
        P[u][v] = u

print(*D,sep = "\n")       #*D 是"解封参数列表"传递参数方式
                           #sep= "\n"表示每个输出元素之间的分割符是换行符"\n"
print()
print(*P,sep = "\n")
print()

for w in range(0,n):                                    #绕道顶点 w
    for u in range(0,n):
        for v in range(0,n):
            #其他顶点 u、v 会不会因为绕道 w 距离变得更短呢?
            if w!=u and w!=v and D[u][w] + D[w][v] < D[u][v] :
                D[u][v] = D[u][w] + D[w][v]
                P[u][v] = P[w][v]

print(*D,sep = "\n")
print()
print(*P,sep = "\n")
```

输出结果：

```
[0, 2, 6, 4]
[100000.0, 0, 3, 100000.0]
[7, 100000.0, 0, 1]
[5, 100000.0, 12, 0]

[0, 0, 0, 0]
[1, 1, 1, 1]
[2, 2, 2, 2]
[3, 3, 3, 3]

[0, 2, 5, 4]
[9, 0, 3, 4]
[6, 8, 0, 1]
[5, 7, 10, 0]

[0, 0, 1, 0]
[3, 1, 1, 2]
[3, 0, 2, 2]
[3, 0, 1, 3]
```

根据路径矩阵 P，对于任何一对顶点 u、顶点 v，其路径可以从终点倒过来追踪到起点，即终点是 v，其前一个顶点是 P[u][v]，再前一个顶点是 P[u][P[u][v]]、...

```
for u in range(0,n):
    for v in range(0,n):
        if u==v continue
        print(u,'到',v,'的逆向路径是:',end=' ')
        print(v,end= ',')
        w = P[u][v]
        while w!=u:
            print(w,end= ',')
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                w = P[u][w]
            print(u)
```

输出结果：

```
0 到 1 的逆向路径是:1,0
0 到 2 的逆向路径是:2,1,0
0 到 3 的逆向路径是:3,0
1 到 0 的逆向路径是:0,3,2,1
1 到 2 的逆向路径是:2,1
1 到 3 的逆向路径是:3,2,1
2 到 0 的逆向路径是:0,3,2
2 到 1 的逆向路径是:1,0,3,2
2 到 3 的逆向路径是:3,2
3 到 0 的逆向路径是:0,3
3 到 1 的逆向路径是:1,0,3
3 到 2 的逆向路径是:2,1,0,3
```

2.9 习题

1．请说明下面哪些是 Python 整型数，哪些是浮点数。

```
11, 17.5, -39, -2.3, 0.12e4, -3.141759, 0.57721566, 7.5e-3
```

2．请说明当在解释器输入 1e1000 或–1e1000 是显示的值是什么，其含义是什么，type(1e1000)的结果是什么。

3．请说明如何检查 None 的类型。

4．请说明下列那句代码会抛出错误。

(A) "I have " + str(5) + " books"

(B) "I said " + ("Hello " * 2) + "1 world"

(C) True + False

(D) "The correct answer to this question is: " + 2

5．为了提高程序效率，对于不可修改的整数对象，Python 解释器开始执行时就为这些对象分配了内存空间，不会被删除回收，因此，多个相同值的整数可能对应的是同一个对象。相同值的整数是否共享同一个对象取决于你解释器。例如，在 jupyter notebook 解释执行时，[–5,256]范围的整数都会共享同一个对象。请在 jupyter nbotebook 和操作系统命令行执行下列 Python 代码，查看输出结果。

```
print(id(100))
print(id(100))
print(id(1000))
print(id(1000))
a=100
b=100
c=1000
d=1000
print(a==b)
print(a is b)
print(c==d)
print(c is d)
```

6．(选做题) sys 模块(模块概念将在第 3 章中介绍)的函数 getsizeof()可查询一个对象占用内存大小。例如：

```
import sys                  #导入模块 sys
sys.getsizeof(3.14)         #调用 sys 模块中的 getsizeof 查询 3.14 占据内存的字节数
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
24
```

模仿这个程序，在该函数中输入整数 3、'h'、'hello'、'hello world'，观察输出结果。

7. 定义两个变量 radius 和 area，分别表示圆的半径和面积，其中 radius 有一个初始值，而 area 的值通过圆的面积公式计算，然后输出它们的值。
8. 用变量 savings 表示存款，用变量 interest 表示年利息(如 0.05)，用变量 growth_multiplier 表示年增长倍数(1+interest)，并用指数运算**计算 8 年后的存款总额保存在结果变量 result，最后输出表示存款总额的值 result。
9. 下面哪些是不合法的标识符？为什么？(　　)

```
@name    def    3re   user_1name  pass-word   _x   234
```

10. 下列代码中有几个变量？几个对象？输出结果是什么？(　　)

```
a = 100
b = a
a= 101
print(b)
```

11. list 是一个复合数据类型，其中的元素可以是任何对象，对于下面的 list 对象 alist。

```
s = 'he'
pi = 3.14
alist = [1,[2,3],s,True, pi]
```

解决下列的五个问题。

(1)输出最后一个元素。

(2)输出最后两个元素。

(3)在最后一个元素前面插入字符串'py'。

(4)将第三个元素替换为 list 对象[4,5]。

(5)删除前两个元素。

12. 指出下列代码中的错误，并说明原因。

```
a=(1,3.14,'hello',{2,3} )
a[2] = 'hello'
b = {1,3.14,'hello',{2,3}}
b[0] = 2
c = {{2,3}:'hello','hello':{2,3}}
```

13. 首先说明下列语句的输出结果是什么，然后将双引号表示的字符串换成单引号表示的字符串，再判断输出结果是什么。

```
print("Yes!\tI\'m a student .\n"
```

14. 给定变量 x 和 y，使用字符串格式打印出 x 和 y 及其总和的值。例如，如果 x = 5 且 y = 3，则输出结果应该是：5 + 3 = 8。
15. 请说明如何检验(25,)表示的是一个 tuple，而(25)表示的是一个整数。
16. 每个学生的信息包括姓名和分数，用姓名作为键，分数作为值，用一个 dict 存储一组学生数据，并通过键值修改或输出某个学生的分数。
17. 请说明下列代码有哪些错误，原因是什么。

```
a = (2, 3.14, 'python', [4,5])
b = {2, 3.14, 'python', [4,5]}
a[3][0] = 'he'
a[1] = 'abc'
b[3][0] = 'he'
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print(a[-5])
print(a[-3])
print(b[3])
```

18．编写程序，要求从键盘输入矩形的长和宽，并输出矩形面积。

19．请说明下列 print()语句有没有错误，如果没有错误，则请判断输出结果是什么。

```
print(3.14/2+True)
print(2*'hello')
print(2+'hello')
```

20．IP 地址是由四个整数构成的一个字符串，如“10.3.9.12”。编写一个程序，从键盘输入一个 IP 地址，再按照下列步骤将它转化为一个整数并输出。

(1)将字符串表示的 IP 地址的每个整数转化为二进制字符串。例如：

```
10 00001010
3 00000011
9 00001001
12 00001100
```

(2)拼接为一个更长的二进制字符串。例如：

```
00001010 00000011 00001001 00001100
```


(3)将这个二进制字符串转换为整数。

21．下列代码可以将一个整数转为一个 IP 地址的四个整数。运行这段代码，并将其改写为从键盘输入一个整数，输出一个形如“10.3.9.12”的字符串形式的 IP 地址。

```
a = 123456789
a,b = a%256,a//256
b,c = b%256,b//256
c,d = c%256,c//256
print(d,c,b,a)
```

22．请说明整数除法和浮点除法有什么区别，整数除法和浮点除法的运算符分别是什么。

23．请分析以下操作的结果是什么，并给出解释。

```
1.5 + 2
1.5 // 2.0
1.5 / 2.0
1.5 ** 2
```

24．请分析下列运算的结果是什么，并给出解释。

```
15 + 20 * 3     13 // 2 + 3     31 + 10 // 3
20 % 7 // 3     2 ** 3 ** 2
```

25．请分析执行下面的语句会发生什么，并给出解释。

```
1 // 0
```

26．判断下列计算 $\frac{b+c}{2a}$ 的语句是否有错误，如有错误请改正。

```
b+c/2a
```

27．请分析下列类型中哪些类型是可变的(mutable)，哪些是不可变的(immutable)，并说明验证方法。

```
list，dict，set，bytearray，int，float，complex，string，tuple，frozenset，bytes
```

28．请分析下列代码有什么错误，并给出解释。

```
a=(1,2,3,[4,5,6])
a[0] = 10
a[3] = 30
a[3][2] = 'he'
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

29．请分析下列代码有什么错误，并请修改正确。

```
number = input("请输入一个数：")
print(type(number))
print(number+30)
```

30．请将下列代码中不正确的缩进修改正确。

```
if score < 60:
print('不及格')
else:
print('及格')

i=0
while i<10:
print(i)
```

31．根据“四年一闰，百年不闰，四百年再闰”，即“闰年是年份能被 4 整除且不能被 100 整除，或者能被 400 整除”这一规则，编写一个程序用于判断一个年份是否为闰年。

32．编写一个程序用于从键盘输入一个人的身高和体重，并根据 BMI 指数公式，判断此人是否肥胖，最后输出判断的结果。

BMI 指数=体重÷身高的平方

根据 BNI 指数判断一个人肥胖的公式是：

低于 18.5：过轻；

18.5～25：正常；

25～28：过重；

28～32：肥胖；

高于 32：严重肥胖。

33．编写一个程序，用于从键盘输入摄氏温度，并将其转为华氏温度并输出。

34．从键盘输入一个一元二次方程的系数，根据不同情况(无根、一个根、两个根)输出不同的信息。

35．编写一个程序用于打印如下金字塔图案，要求行数从键盘输入。

```
    *
   * *
  * * *
 * * * *
```

36．编写一个程序用于从键盘输入一个弧度 x，以计算正弦函数 sin(x)的值。要求最后一项的绝对值小于 10^{-5}，并统计出此时累计了多少项。sin(x)的近似计算公式为：

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \ldots$$

37．模仿整数之和的程序，将统计学生平均成绩的程序改写成用 break 语句终止循环体的执行。

38．编写一个程序，用于判断从键盘输入的一个整数是否为质数。

39．猜数字游戏。假设程序中设置了一个 1 到 100 之间的正整数 num，用户从键盘输入一个猜想的数字 guess，如果 guess 等于 num，那么就显示祝贺用户成功的信息，如果 guess 不等于 num，就提示用户继续输入，直到超过指定的猜测次数(如 8 次)就提示用户失败信息。提示，核心程序如下所示。

```
num = 37
i = 0
success = 0            #是否成功的标志
while i<8:
    guess= int(input("请输入猜测的数字："))
    ?
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
if(success)
    print("很遗憾，猜测失败！")
continue
```

40．编写一个程序，用于计算 1 到 1000 之间所有的奇数之和。

41．请去掉第 39 题“猜数字游戏”中的 success 标志变量，并使用循环语句的 else 子句改写“猜数字游戏”。

42．编写一个程序，用于对一个班级中所有学生的某门课进行成绩分析。一个学生的信息包括学号、姓名、平时成绩、实验成绩、期末成绩、总评成绩。总评成绩是平时成绩、实验成绩和期末成绩的加权平均。学生成绩数据、成绩的加权系数等均需要从键盘输入。程序包括成绩录入、修改、查询、加权系数的查询和修改、计算总评成绩、统计不同分数段的人数和百分比、最高分和最低分之差等。程序需一直运行，直到用户输入一个特殊的值才可退出。

以下是一个核心框架，可在此基础上编写代码。

```
print('请输入代表不同命令的字母：')
help_info =('''
    'i' or 'I':'录入一个学生信息'
    'q' or 'Q':'按姓名查询学生信息'
    'm' or 'M':'按学号修改学生信息'
    'w' or 'W':'显示/修改加权系数'
    'c' or 'c':'计算所有学生的总评、统计信息'
    's' or 'S':'显示所有学生信息'
    't' or 'T':'显示统计信息'
    'x' or 'X':'显示统计信息'
    ''')
print(help_info)
student_infos = []
while True:
    cmd = input('请输入一个命令：')
    if cmd=='x' or cmd=='X': break
    elif cmd=='i' or cmd=='I':
        id = input('请输入学号：')
        name = input('请输入姓名:')
        score_ = float(input('请输入平时成绩:'))
        score_exp = float(input('请输入实验成绩:'))
        score_test = float(input('请输入期末成绩:'))
        student_infos += [[id,name,score_,score_exp,score_test,0]]
    elif cmd=='s' or cmd=='S':
        #print(student_infos)
        for s in student_infos:
            print(s)
    pass
```

也可以用 list 和 str 类型的方法简化上述的代码。例如：

- 用一个 list 对象（上述代码中的 student_infos）存储所有学生的信息，可以用 list 的函数 append()在最后添加一个元素，如：

```
student_infos.append([id,name,score_,score_exp,score_test,0])
```

- 用 str 的函数 split()将输入的字符串分割成多个字符串。当然，也可以逐个调用函数 input()。例如：

```
id,name,score_,score_exp,score_test =
    input("请输入:学号、姓名、平时成绩、实验成绩、期末成绩：").split(' ')
score_ = float(score_)
score_exp = float(score_exp)
score_test = float(score_test)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第3章 函　　数

函数(function)就是一个**命名**的**程序块**。正如变量是为一个对象起一个名字一样(变量引用对象)，函数就是给一组语句起了一个名字，或者说，函数名引用了这组语句(**程序块**)。通过这个函数名，就可以多次**调用执行**这个程序块中的语句。前面章节中的代码已多次通过函数名执行函数名对应的程序块语句，如通过内置函数 print()执行向屏幕窗口输出信息的功能。利用函数可以使代码只要编写一次，就可以多次调用，从而使代码具有复用性。

3.1 定义函数、调用函数、参数传递

3.1.1 定义函数和调用函数

Python 通过关键字 **def** 定义函数(给 1 个程序块起 1 个名字)。例如，下列代码定义了 1 个叫作 hello 的函数：

```
def hello():
    print("hello,")
    print("world")
```

该函数的程序块中包含了 2 条函数 print()调用语句。此后，就可以通过函数名 hello 调用这个函数 hello()。例如，下面代码调用了 2 次函数 hello()：

```
hello()
hello()
```

输出结果：

```
hello,
world
hello,
world
```

假如希望上述的函数 hello()可以根据不同的输入，如不同的人名，输出针对特定人名的输出信息，为此可以进行如下修改：

```
# hello 函数包含一个叫作 name 的参数
def hello(name):
    print("hello,")
    print(name)
```

函数的函数名后的圆括号里说明了一个参数 name，函数体的 print(name)就输出这个 name 参数的值。调用该函数时，就可以向这个 name 传递不同的对象。例如：

```
hello("Li Ping")          #将"Li Ping"传递给被调用函数 hello()的 name 参数
hello("Zhang Wei")
```

例如，语句 hello("Li Ping")调用函数 hello()，并将“Li Ping”传递给函数 hello()的 name 参数，即 name 引用的就是这个字符串“Li Ping”。函数 hello()的 print(name)将输出这个 name 引用的“Li Ping”。

因此，定义函数的语法格式如下：

```
def 函数名(参数 1，参数 2，…):
    函数体程序块
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

def 关键字之后是函数名，函数名后是一对圆括号，圆括号中可以包含一些称为“**参数**”的变量，圆括号中包含的参数变量称为函数的“**参数列表**”(参数列表可以是空的，即不包含任何参数)，圆括号的后面是一个冒号：。从 def 关键字一直到该行结尾的冒号构成了**函数头**。函数头下面的是一组语句构成的语句块，称为**函数体**。

因此，函数由**函数头**和**函数体**构成。函数头说明了函数的名字和参数有哪些，而函数体是具体的函数代码(语句)。函数的参数说明了函数的调用者可以向该函数传递什么样的数据，这些参数之间用逗号隔开。函数也可以没有任何参数，如同上面的函数 hello()一样，但没有参数的函数总是执行同样的数据处理，不能根据实际情况执行不同的数据处理。

调用函数就是通过传递一些实际数据给函数后圆括号中的“参数”，从而调用执行这个函数名引用的函数体语句，即执行这个函数中的代码语句。

```
s=input("请输入一个人名:")
hello(s)
```

其中，第 2 句调用了函数 hello()，并传递从键盘输入的字符串 s 给函数 hello()的参数 name。

输出结果：

```
请输入一个人名:李平
hello,
李平
```


将键盘输入的字符串“李平”传给了函数 hello()的 name 参数，函数 hello(name)的“print(name)”语句将输出参数 name 指向的对象，也就是 s 指向的字符串对象“李平”。

如果调用函数 hello()时没有提供参数，例如：

```
hello()
```

则会报错，错误提示是“缺少一个需要的位置参数：‘name’”：

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-a75d7781aaeb> in <module>()
----> 1 hello()

TypeError: hello() missing 1 required positional argument: 'name'
```

如果调用函数 hello()时提供了 2 个以上的参数，则同样会报错，例如：

```
hello('hello','world')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-fac44797b17f> in <module>()
----> 1 hello('hello','world')

TypeError: hello() takes 1 positional argument but 2 were given
```

3.1.2 参数传递

定义函数时，圆括号的参数称为“**形式参数**(parameters)”，又称形参，而调用函数时传递的数据称为“**实际参数**(arguments)”，又称实参。调用函数时，将实际参数传递给形式参数的过程称为“**参数传递**”。例如，上面的代码将实际参数 s 传给形式参数 name，实际上这是一个实际参数给形式参数赋值的过程，参数传递相当于执行了：

```
name = s
```

因此，参数传递就是给形式参数赋值，和普通的变量赋值没有任何区别。例如：

```
def fun(x):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    print("x 的 id:", id(x))
    x+=3
    print("x 的 id:",id(x))
    print("x 的值:",x)

a = 2
print("a 的 id:",id(a))
fun(a)
print("a 的值:",a)
print("a 的 id:",id(a))
```

输出：

```
a 的 id: 1717202432
x 的 id: 1717202432
x 的 id: 1717202528
x 的值: 5
a 的值: 2
a 的 id: 1717202432
```

当调用 fun(a)时，将 a 赋值给形参 x，这时实参和形参是同一个对象的引用(如图 3-1(a)所示)。但当执行 x+=3 时，也就是执行 x= x+3 时，为 x+3 创建了一个新对象，形参 x 引用了这个新的对象(如图 3-1(b)所示)，其后一句的 print("x 的 id", id(x))打印的就是新对象的 id。接着的 print("x 的值:",x)就是打印的这个 x 指向的新对象的值 5。当这个函数结束后，print("a 的值:", a)打印的仍然是 a 指向的原来的对象(值 2)。

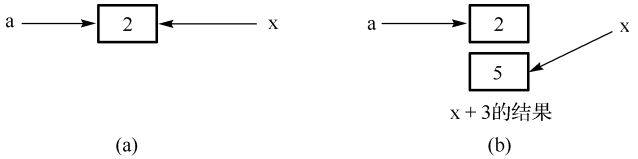


图 3-1　形参引用实参、给形参赋值

再看以下例子：

```
def fun_2(x):
    print("x 的 id:", id(x))
    x[0] = 3.14
    print("x 的 id:", id(x))
    print("x 的值:", x)

a = [2,3]
print("a 的 id:", id(a))
fun_2(a)
print("a 的值:", a)
print("a 的 id:", id(a))
```

输出：

```
a 的 id: 2826240248712
x 的 id: 2826240248712
x 的 id: 2826240248712
x 的值: [3.14, 3]
a 的值: [3.14, 3]
a 的 id: 2826240248712
```

因为在函数 fun_2()里只修改了 x 指向的可变对象，而没有修改 x 本身(没有给 x 赋值)，所以 x 始终指向的都是实参 a 指向的对象。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

3.1.3 return 语句

一个函数可以通过关键字 return 返回函数的处理结果。例如，编写一个计算两个整数之间的所有整数之和的函数：

```
def sum(m,n):
    if m>n:                 #保证 m 不大于 n
        m,n = n,m           #交换 m,n
    sum = 0
    i=m
    while i<=n:
        sum += i
        i += 1
    return sum
```

上述函数中的“**return** sum”通过关键字 return 返回 m 和 n 之间整数之和 sum。调用这个函数时，需要传递两个整数给它。

```
ret = sum(3,10)
print(ret)
```

输出：

```
52
```


1．最大公约数

两个正整数的最大公约数就是能被二者整除的最大正整数。给定两个正整数 m 和 n，假设用符号 GCD(m,n)表示二者的最大公约数，它们的最大公约数可以用下列式子来计算：

$$\mathrm{GCD}(m,n)=\begin{cases} m & n=0 \\ \mathrm{GCD}(n,m\%n) & n!=0 \end{cases}$$

其中，%表示取余数运算。因此，可以用如下的迭代的方法求两个正整数(72 和 27)的最大公约数：

$$\begin{aligned}\mathrm{GCD}(72,27)&=\mathrm{GCD}(27,72\%27)=\mathrm{GCD}(27,18)\\&=\mathrm{GCD}(18,27\%18)=\mathrm{GCD}(18,9)\\&=\mathrm{GCD}(9,0)=9\end{aligned}$$

根据上述过程，可以定义一个求最大公约数的函数。例如：

```
def GCD(m,n):
    while n!=0:
        m, n = n, m%n
    return m
```

这个函数 GCD()用于计算两个形参表示的整数的最大公约数，通过关键字 return 返回结果。有了这个函数，其他的程序代码就可以多次调用这个函数来计算两个整数的最大公约数。例如：

```
m=72
n=27

gcd = GCD(m,n)
print('gcd(72,27):=',gcd)

m = 24
n = 36
gcd = GCD(m,n)
print('gcd(24,36):=',gcd)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
gcd(72,27):= 9
gcd(24,36):= 12
```

2．返回多个值

关键字 return 不仅可以返回一个值，还可以同时返回多个值，这些值将会被包裹在一个 tuple 对象中返回。例如，下列的代码可以返回一个 list 对象中所有数的最大值和最小值。

```
def min_max(arr):
    m= arr[0]
    M= arr[0]
    for e in arr:
      if e<m:
         m = e
      if e>M:
        M = e
    return m,M;
ret = min_max([36,2,19,76,91,47])
print(ret)
print(ret[0])
print(ret[1])
```

输出：

```
(2, 91)
2
91
```


3.1.4 文档字符串

定义函数时，可以在函数头后面添加由三个引号(三个单引号或三个双引号)括起来的文档字符串(docstring)，用于说明这个函数的功能。docstring 会作为函数对象的一个属性“__doc__”被使用。例如，可以通过函数 print()输出这个属性：

```
def hi():
    '''
    hi,
    这是一个 doc string 的例子
    '''
    print("hi,world")
```

使用这个函数：

```
hi()                    #调用 hi 函数
print(hi.__doc__)       #打印 hi 函数的 __doc__属性
```

输出：

```
hi,world
hi,
    这是一个 doc string 的例子
```

文档字符串可以有助于他人理解这个函数的功能，Python 的每个对象都有一个“__doc__”属性，除直接输出一个对象的“__doc__”属性外，Python 的内置函数也同样可以输出文档字符串的内容。许多文档自动化工具可以从程序代码中提取文档字符串，用于进行处理、打印、显示文档等操作。

总结

- 函数定义包含函数头和函数体。函数头以关键字 **def** 开头，后面跟函数名，函数名后是圆括

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

号，圆括号里可以包含参数(圆括号里的所有参数称为函数的**参数列表**)，圆括号后是一个冒号。函数体里的语句必须具有一致的缩进。

- 调用函数的方法：函数名后用圆括号传递实参给形参。
- 实参传递给形参的参数就是普通的变量赋值，也就说，形参和实参引用的是同一个对象(当实参本身是对象时，形参就是这个实参对象的引用)。
- 函数中对形参的赋值使形参指向了另外的对象，但对形参引用的可变对象的修改不会改变形参本身。
- 函数通过关键字 **return** 返回计算的结果。
- 函数头后面可以通过 3 个引号表示的字符串给函数添加一个文档串属性**__doc__**。这个属性是对函数的说明，可以帮助使用者通过查询这个文档串了解这个函数的功能和使用方法。

3.2 全局变量和局部变量

3.2.1 全局变量

函数外部(或称全局作用域，global scope)声明的变量，称为**全局变量**(global variable)，这意味着全局变量在函数内部或外部都可以使用。例如：


```
x = "global"
def fun():
    print("x inside :", x)

fun()
print("x outside:", x)
```

输出：

```
x inside : global
x outside: global
```

如果试图在一个函数里修改一个全局变量。例如：

```
x = "global"
def fun():
    x = x*2
    print(x)

fun()
```

则将产生错误(提示：“局部变量在赋值前被引用”的错误)：

```
---------------------------------------------------------------------------

UnboundLocalError       Traceback (most recent call last)

<ipython-input-3-ce8c348bb019> in <module>()
     5    print(x)
     6
----> 7 fun()
<ipython-input-3-ce8c348bb019> in fun()
     2
     3 def fun():
----> 4    x = x*2
     5    print(x)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
      6
UnboundLocalError: local variable 'x' referenced before assignment
```

由上例可知，函数 fun()内部使用形如“x=”的语句就相当于定义了**局部变量**(local variable)，因此，这个函数里的 x 就不是全局变量的那个 x 了，而等号的右侧的“x*2”试图使用这个还未定义的局部变量 x，所以产生了上述错误。

那么，如何在函数内部对全局变量修改呢？方法是使用关键字 **global** 声明 x 是一个全局变量。例如：

```
def fun():
    global x
    x = x*2
    print(x)

fun()
```

输出：

```
globalglobal
```

3.2.2 局部变量

可以在函数内部定义一个变量，这个变量只属于该函数，外部无法使用这个变量，该变量称为**局部变量**(local variable)。例如：

```
def fun():
    y = "local"

fun()
print(y)       # 错：试图访问 fun()函数的局部变量
```

执行后将产生“名字 y 没有定义”的错误：

```
---------------------------------------------------------------------------

NameError                 Traceback (most recent call last)

<ipython-input-6-3facf76a0729> in <module>()
      3
      4 fun()
----> 5 print(y)                #错：试图访问 fun 函数的局部变量
NameError: name 'y' is not defined
```

如果函数内部声明的局部变量和外部变量同名，则内部的局部变量会隐藏全局变量，即在函数内部使用这个名字访问的总是局部变量。例如：

```
x = 5                       #全局变量 x
def fun():
    x = 10                  #定义了一个新的局部变量 x
    print("local x:", x)

fun()
print("global x:", x)
```

输出：

```
local x: 10
global x: 5
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

总结

- 函数内部可访问但不能直接修改全局变量，必须用关键字 global 声明才能修改全局变量。
- 局部变量和外部变量同名，在函数内部通过这个名字访问的都是局部变量，除非用 global 声明为全局变量。函数外部不能访问局部变量。

3.3 函数的参数

3.3.1 默认形参

函数的形参可以有默认值，称为“默认形参”，调用函数时如果没有为默认形参提供实际参数，则该形参就取默认值。例如：

```
def date(year, month='01',day = '01'):
   print(year,month,day)

date(2018)
date(2018,'07')
date(2018,'07','25')
```


输出：

```
2018 01 01
2018 07 01
2018 07 25
```

如果一个函数的形参中既有默认形参也有非默认形参，则默认形参必须都在非默认形参的后面，默认形参后面不能再有非默认形参。例如：

```
def f(a,b= 2,c):
   pass
```

执行后产生语法错误（“默认形参 b 的后面又出现了非默认形参 c”）：

```
  File "<ipython-input-1-dcdf3a460ecb>", line 1
    def f(a,b= 2,c):
          ^
SyntaxError: non-default argument follows default argument
```

默认形参的默认值是函数定义时就计算好的。例如：

```
i = 5
def f(arg=i):
   print(arg)

i = 6
f()   # 将输出:5
```

输出：

```
5
```

因为定义函数 f()时，默认形参 arg 的值等于 i(i=5)，尽管 i 变量之后修改了，但并不会改变最初的函数 f()的默认值，所以输出是 5。

由于默认形参的默认值只在函数定义时计算一次，所以每次函数调用这个默认形参时，始终指向的都是初始化的那个对象。如果这个对象是一个可变对象(mutable object)，则当每次函数调用时，如果对这个默认形参引用的这个对象进行修改，则修改的将都是同一个对象。例如：

```
def f(var, arr=[]):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    arr.append(var)
    return arr

print(f(1))
print(f(2))
```

输出：

```
[1]
[1, 2]
```

上例中函数定义时，arr 引用的是空的 list 对象，函数调用 f(1)、f(2)中的 arr 也始终引用的是同一个 list 对象，在 f(1)中将 1 追加到这个 list 对象(arr.append(1))，在 f(2)中将 2 追加到这个 list 对象(arr.append(2))。

如果希望每次调用函数时默认形参指向的是不同的对象，则可以采用下面的技巧：

```
def f(var, arr=None):
    if arr==None:
        arr=[]
    arr.append(var)
    return arr

print(f(1))
print(f(2))
```


输出：

```
[1]
[2]
```

默认形参 arr 初始化为 None 对象，每次调用函数 f()时默认形参 arr 都是引用这个对象，但函数体开头因为满足条件“arr==None”，所以对 arr 的重新赋值“arr=[]”使得这个 arr 指向了一个新的空的 list 对象[]，而不再是原先参数 arr 引用的那个 None 对象了。实际上，Python 可以通过赋值运算符重新定义同名的变量以引用不同的对象，即每个函数 f()中的 arr 执行过 if 语句后，就引用一个新的空的 list 对象[]。

也可以去掉这个默认形参。例如：

```
def f(var):
    arr = []
    arr.append(var)
    return arr

print(f(1))
print(f(2))
```

输出：

```
[1]
[2]
```

当然，还可以为默认形参传递一个实参，此时，形参就与这个实参引用同一个对象。

3.3.2 位置实参和关键字实参

函数定义中的形参是有顺序的，调用函数时传递的实参是按照顺序为对应位置的形参赋值的。这种按照位置顺序传递的实参称为“**位置实参**”。例如：

```
def hello(name, msg = "Good morning!"):
   print("哈罗! ",name + ', ' + msg)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
#调用函数：传递位置参数
hello("小白")                  #"小白"作为第 1 个位置实参传递给形参 name，形参 msg 有默认值
hello("老张","你好吗?")        #"老张"作为第 1 个位置实参传递给形参 name
                               #"你好吗?"作为第 2 个位置实参传递给形参 msg
```

输出：

```
哈罗！ 小白, Good morning!
哈罗！ 老张, 你好吗?
```

另外还有一种称为“**关键字实参**”的参数传递方式，该传递方式在传递实参时指明这个实参传递给哪个形参。其语法格式是：

```
形参名=实参
```

也可以通过**关键字实参**传递实参给被调用函数。例如：

```
hello(name = "小白",msg = "你好吗?")   #形参 name=实参"小白"，形参 msg=实参"你好吗"
hello(msg = "你好吗?",name = "老张")   #形参 msg=实参"你好吗"，形参 name=实参"老张"
# 1 个位置参数，1 个关键字参数
hello("李平",msg = "你好吗?")
```

输出：

```
哈罗！ 小白, 你好吗?
哈罗！ 老张, 你好吗?
哈罗！ 李平, 你好吗?
```


无论函数的形参是否有默认值，都可以采用“位置实参”或“关键字实参”的方式将实参传递给形参。

3.3.3 任意形参（可变形参）

如果在定义函数时不知道将来使用者调用这个函数时传递的实际参数个数，则可以在定义函数时，在这个形参名前加一个星号*。函数定义中的这种形参称为“**任意形参**”或“**可变形参**”。传递给可变形参的多个实参被组装成一个 tuple 对象并传递给这个可变形参。

例如：

```
def hello(*names):          #可变形参 names 可以接收任意多的实参
  print("哈罗:")
  for name in names:        #对比可变形参 names 元组(tuple)的每个元素 name
     print(name)
  print();
```

如果调用这个函数：

```
hello("小白","老张")          #给 names 形参传递两个实参
hello("小白","老张","老王") #给 names 形参传递三个实参
```

输出：

```
哈罗:
小白
老张

哈罗:
小白
老张
老王
```

在函数调用时，可以传递多个实参给可变形参，这些实参(如“小白”“老张”)都被打包为一

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

个 tuple 对象传递给函数的可变形参，函数内部可以用“**for...in**”或**下标运算符**[]访问该可变形参引用的 tuple 对象中的元素。

如果函数的形参中既有默认形参又有可变形参，则默认形参必须位于可变形参的后面。例如：

```
def date(*args, sep="/"):
   return sep.join(args)

print(date("2018", "07", "25"))      #给 args 可变形参传递了 3 个实参
date("2018", "07", "25", sep=".")
```

输出：

```
2018/07/25
'2018.07.25'
```

注意

函数定义中的可变形参最多只能有一个，不能有两个可变形参，否则，如果给这些形参传递实参，就无法区分哪个实参是传给哪个形参了。

3.3.4 字典形参

和可变形参类似，还有一个**字典形参**。字典形参名前面有 2 个**，这个形参指向的是一个 dict 对象，调用函数时必须以“key=value”的形式传递可变数量的实参，这些实参被组装成一个 dict 对象，并赋值给字典形参。

如果函数定义中既有**可变形参**又有**字典形参**，则**字典形参**必须位于**可变形参**的后面。例如：

```
def f(x, *y, **z):
    print("x:",x)
    #访问任意形参中的参数
    for e in y:
        print(e)
    print()
    #访问字典形参中的参数
    for key in z:
        print(key, ":", z[key])
```

如果调用这个函数：

```
f("hello", "li ping",60.5, year="2018",month=7,day=25)
```

则输出：

```
x: hello
li ping
60.5

year : 2018
month : 7
day : 25
```

在调用函数时，通过“key=value”（“键-值”）的形式（“year="2018",month=7,day=25”）将实参传给字典形参。而在函数体中，则是通过字典形参获取传来的这些实参“键-值”。也就是说，这些实参被打包到了字典形参中。

注意

函数定义中的字典形参最多只能有一个，不能有两个字典形参，否则，如果给这些形参传递字典实参，则无法区分哪个字典实参是传给哪个字典形参了。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

3.3.5 解封参数列表

假设有一个如下的函数用于计算两个对象(数)的和：

```
def add(x,y):
   return x+y
```

调用这个函数时，就必须传递两个实参给它：

```
print(add(3,5))
```

输出：

```
8
```

假设需要求和的两个数放在一个 tuple 或 list 对象中，能否将这两个数直接传递给这个函数 add()呢？例如：

```
ab = [3,5]
print(add(ab))
```

执行后产生 TypeError(类型错误)(“缺少一个位置参数”)的错误：

```
---------------------------------------------------------------------------

TypeError                    Traceback (most recent call last)

<ipython-input-14-a219a64a643d> in <module>()
     1 ab = [3,5]
----> 2 print(add(ab))
TypeError: add() missing 1 required positional argument: 'y'
```

以上错误是因为函数需要两个实参，而这里只传递了一个实参 ab。解决方法是，通过下标运算符从这个 ab 对象中获得两个实参。例如：

```
print(add(ab[0],ab[1]))
```

输出：

```
8
```

有没有一种更方便的方法呢？答案是：有。将这个 list 或 tuple 变量名前用一个*作为实参传给被调用函数。Python 解释器会自动从这个 list 或 tuple 对象中解析出每个实参并传递给被调用函数。这种传递实参的方式称为“**解封实参列表**”。例如：

```
print(add(*ab))
```

输出：

```
8
```

再如，生成“两个整数之间的一系列整数的可迭代对象”的函数 range()需要单独的 start 和 end 参数，而 start 和 end 的值如果在一个 list 或 tuple 对象中，就可以通过这种解封实参列表方法将 list 或 tuple 对象的数据元素作为实参传递给函数 range()。例如：

```
#函数 range 接收两个参数：range(start,end)
s = list(range(3,7))
print(s)
args = [3, 7]
s = list(range(*args))          #*args 将 args 列表中的元素 3 和 7 分离出来
print(s)
```

输出：

```
[3, 4, 5, 6]
[3, 4, 5, 6]
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

类似地，假设参数在一个字典中，则要用两个星号**将它们分离出来。例如：

```
def f(name, score = 0.0):
  print('the name: ',name)
  print('the score:',score)
d = {"name": "li ping", "score": 60.5}
f(**d)   #**d 将字典中的参数分离出来
```

输出：

```
the name:  li ping
the score: 60.5
```

总结

- 函数的形参可以有默认值，称为**默认形参**，形参名前有一个*的称为**可变形参**，形参名前有两个**的称为**字典形参**。可变形参必须在非默认形参的后面，默认形参必须在非默认形参和可变形参的后面，字典形参必须放在最后面。
- 函数定义中的形参是有顺序的，实参可以按照位置传递给形参，称为**位置实参**，也可以按照**形参名=实参**的方式将实参传递给形参，称为**关键字实参**。关键字实参可以任意顺序排列。
- 可以给可变形参传递多个实参，这些实参被打包成一个 tuple 对象传递给可变形参。函数可以像普通 tuple 对象一样访问可变形参中的实参。
- 可以采用**键-值**的方式将字典实参传递给字典形参。这些键-值实参被打包成一个字典对象传给字典形参。函数可以像普通字典对象一样访问字典形参中的每个键-值实参。
- 假如要传递给函数的实参放在一个 tuple 或 list 对象中，则可以通过在指向这个对象的变量名前加*的**解封实参列表**方式传递给被调用函数，list 或 tuple 中的这些实参将被解封传递给被调用函数的形参。假如要传给函数的实参放在一个 dict 对象中，则可以通过在指向这个对象的变量名前加**的**解封实参列表**方式将字典实参传递给形参。

3.4 递归函数(调用自身的函数)

3.4.1 递归函数的使用方法

1. 什么是递归函数

由上述讲解可知，一个函数可以在其内部调用其他函数。如果一个函数在其内部存在调用该函数自身的语句，就称为**递归函数**。

递归是一种将任务分解的解决问题的方法，一个大的问题如果能够分解成和它类似的子问题，且子问题的解决方法和大问题一样，只不过问题的规模有所区别而已，则这种情况就可以采用递归的方法来解决这个问题。

例如，求一个 n 的阶乘问题，可以通过 n 和 n−1 的阶乘相乘而得到，即问题规模为 n 的阶乘问题分解成了规模更小的 n−1 的阶乘问题，即 n!=n×(n−1)!

因此，可以编写下面的代码来求 n 阶乘：

```
def fact(n):
    if n==1:                    #如果 n 等于 1，就直接返回值 1
        return 1
    return n * fact(n - 1)      #如果 n 大于 1，就是 n 和 fact(n-1)的乘积
fact(4)                         #输出： 24
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
24
```

当 n=1 时，就不需要再分解了，即不需要再递归为更小的子问题了。这种不需要再分解的问题，即“规模是 n=1 的阶乘问题”称为“**基问题**”。

递归是一个嵌套的过程。例如，fact(4)的递归计算过程如下：

```
===> fact(4)
===> 4 * fact(3)
===> 4 * (3 * fact(2))
===> 4 * (3 * (2 * fact(1)))
===> 4 * (3 * (2 * 1 ))
===> 4 * (3 * 2)
===> 4 * 6
===> 24
```

2．斐波那契数列

斐波那契数列(Fibonacci sequence)，又称黄金分割数列，因数学家列昂纳多·斐波那契(Leonardoda Fibonacci)以兔子繁殖为例子而引入，故又称“兔子数列”，指的是数列{f(n)|n = 0,1,2,…}。例如：

```
f(0)=1，f(1)=1，当 n>=2 时，f(n)=f(n-1)+f(n-2)
```

可编写如下的递归函数：

```
def fib(n):
    if n<=2 :
        return 1
    else:
        return fib(n-1)+fib(n-2)

for i in range(8):
    print(fib(i),end = ',')
```

输出：

```
1,1,1,2,3,5,8,13,
```

3．最大公约数

最大公约数也是一个递归问题：

$$\gcd(m,n)=\begin{cases} m & n=0 \\ \gcd(n,m\%n) & n!=0 \end{cases}$$

可编写如下的递归函数：

```
def gcd(m,n):
    if n==0 :
        return m
    else:
        return gcd(n,m%n)

print(gcd(72,27))
print(gcd(24,36))
```

输出：

```
9
12
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

3.4.2 实战：二分查找的递归实现

本书第 2 章中的二分查找问题可以看作一个递归问题，在非空的原序列上的查找问题，被分解为三个子问题：(1)和中间的元素的直接比较问题；(2)左区间上的查找问题；(3)右区间上的查找问题。可以编写基于递归的二分查找程序：

```
def binarySearch(alist, value):
    if len(alist) == 0:                                    #(0)空序列
        return -1
    else:
        Middle = len(alist)//2
        if alist[Middle]==value:                           #(1)中间元素直接比较
            return Middle
        else:
            if value<alist[Middle]:
                return binarySearch(alist[:Middle],value) #(2)左区间查找
            else:
                return binarySearch(alist[Middle+1:],value) #(3)右区间查找
```

可以用下列代码测试这个递归二分查找函数：

```
testlist = [5,7,12,25,34,37,43,46,58,80,82,105]
print(binarySearch(testlist, 25))
print(binarySearch(testlist, 13))
```

输出：

```
3
-1
```

3.4.3 实战：汉诺塔问题

汉诺塔问题(如图 3-2 所示)是由法国数学家爱德华·卢卡斯在 1883 年发明的(他的灵感来自一个印度教传说)。假设有三根柱子 a、b、c，其中 a 柱子上有 n 个($n>1$)盘子，盘的尺寸从下到上依次变小。现在要求将盘子全部移到 c 柱子，每次只能移动一个盘子，且小盘必须在大盘之上，当然，盘子只能放在三个柱子之一上。

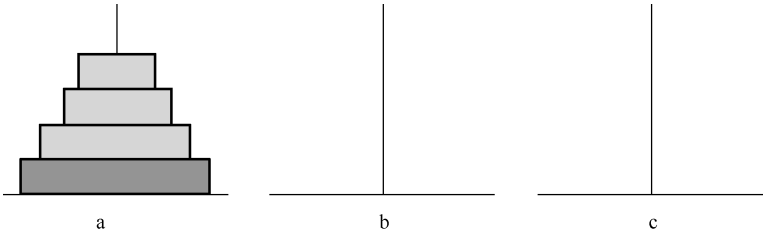


图 3-2 汉诺塔问题

假设采用蛮力尝试法，处于一个柱子上的盘子最多有两个选择(移动到另外两个柱子之一)，n 个盘子都这样尝试，至少需要 n 次尝试，因此至少需要移动 2^n-1 次，但由于每个盘子不可能只尝试一次，所以总的次数会远远大于这个数字。

如果采用“分而治之”的解决问题方法，就可以将“n 个盘子的移动(从 a 柱子借助 b 柱子移到 c 柱子)”分解为如下子问题(如图 3-3 所示)：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

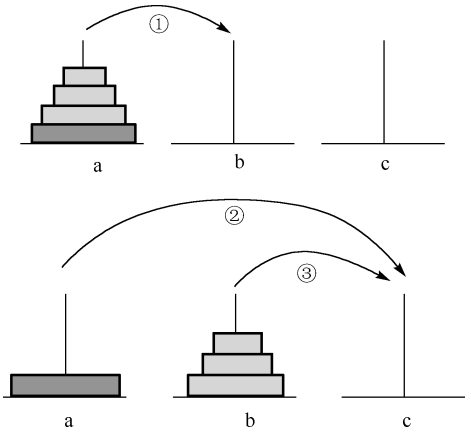


图 3-3　汉诺塔的递归分解

- （上面的）n–1 个盘子的移动（从 a 柱子借助 c 柱子移到 b 柱子）。
- 最大盘子的移动：直接从 a 柱子移到 c 柱子。
- n–1 个盘子的移动（从 b 柱子借助 a 柱子移到 c 柱子）。

当然，对于 n=1 的“基问题”，不需要分解，直接移动即可。因此，可以编写下列代码：


```
#一个盘子：直接移动
def moveDisk(i,x,y):
    print("moving disk",i," from",x,"to",y)

#盘子数，起始柱，中转柱，目标柱
def move(n, a, b, c):
    if n>= 1:
        move(n-1, a, c , b)     #n-1 个盘子从 a 柱子借助 c 柱子移到 b 柱子
        moveDisk(n,a,c)         #第 n 号盘子直接从 a 柱子移到 c 柱子
        move(n-1, b, a, c)      #n-1 个盘子从 b 柱子借助 a 柱子移到 c 柱子
```

下面的代码对 n=3 调用这个 move()函数：

```
move (3,'A','B','C')
```

输出直接移动的步骤：

```
moving disk 1  from a to c
moving disk 2  from a to b
moving disk 1  from c to b
moving disk 3  from a to c
moving disk 1  from b to a
moving disk 2  from b to c
moving disk 1  from a to c
```

3.4.4　实战：快速排序算法

对一组数进行排序的快速排序是一个递归过程。首先在这组数中任意选取一个数作为“基准”，将这组数分为两部分，其中一部分的所有数不大于基准元素，而另外一部分的所有数不小于基准元素。例如，有一组数：

```
34，2，89，47，29，13
```

假设任意选取一个数 34 作为基准元素，然后将这组数按照 34 分为两部分：

```
2，29，13，[34]，89，47
```

将一个序列按基准元素 34 一分为二，其中，左半部分的数都小于等于 34，右边部分的都大于

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

等于 34，这个过程称为“**一次划分**”，对分割好的两部分重复上述过程，如此进行下去，直到每个部分的长度不超过 1。

因此，整个快速排序算法(假设叫作 qsort)分为三个步骤：首先一次划分；再对左、右部分分别调用这个快速排序算法(qsort)。可以编写如下代码：

```
#对[start,end]区间的元素进行快速排序
def qsort(arr, start ,end):
    if start<end:
        pivot=partition(arr,start,end)  #将[start,end]之间的序列一次划分为两部分
                                        #pivot 是基准的位置
        qsort(arr,start,pivot-1)        #对[start,pivot-1]之间的序列调用 qsort 快速排序过程
        qsort(arr,pivot+1,end)          #对[pivot+1,end]之间的序列调用 qsort 快速排序过程
```

递归函数 qsort()里调用了对一个区间完成一次划分的函数 partition()，其过程如图 3-4 所示。假设第一个元素是基准元素，则可以使用首尾两个指示器，当右指示器指向的元素大于等于基准元素时，则该指示器向左移动，否则就停止；当左指示器指向的元素小于等于基准元素时，则该指示器向右移动，否则就停止。当两个指示器都停止时，左、右指示器指向的元素都分别大于或小于基准元素，这时就交换它们指向的元素值，再继续这个“两个指示器向内靠拢”的过程。直到两个指示器相遇，这个位置就是基准元素的目标位置。

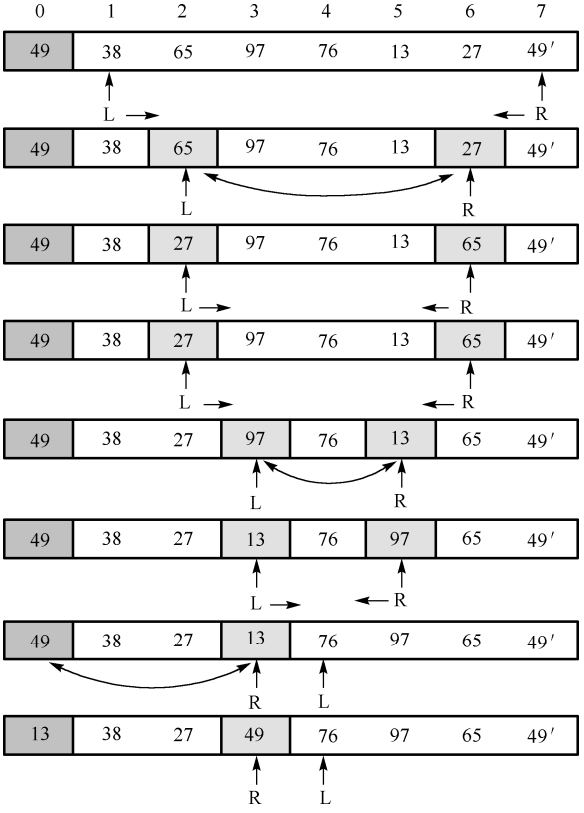


图 3-4 “一次划分”过程

函数 partition()代码如下：

```
def partition(alist,start,end):
   pivotvalue = alist[start]   #假设选择 start 的元素为基准元素，并暂存到 pivotvalue 中
   L = start+1                 #左指示器指向区间左侧
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    R = end                          #右指示器指向区间右侧

    done = False
    while not done:
        while L <= R and alist[L] <= pivotvalue:
            L = L + 1

        while alist[R] >= pivotvalue and R >= L:
            R = R -1

        if R < L:
            done = True
        else:
            alist[L],alist[R] = alist[R],alist[L]
            #temp = alist[L]
            #alist[L] = alist[R]
            #alist[R] = temp
#R<L 时的位置 R 就是基准元素的目标位置
    alist[R],alist[start] = alist[start],alist[R]    #交换基准元素和 R 位置的元素
    return R                 #返回基准元素的位置
```


调用对一个区间的快速排序递归函数 qsort()，可对一个序列进行快速排序。例如：

```
def quickSort(alist):
    qsort(alist,0,len(alist)-1)  #调用递归函数 qsort 对[0,len-1]区间进行快速排序

alist = [49,38,27,97,76,13,27,49]
quickSort(alist)
print(alist)
```

输出：

```
[13, 27, 27, 38, 49, 49, 76, 97]
```

如果不追求效率，还可以采用一种取巧的简单方法：遍历这个数组列表，将其中小于基准元素的数放入一个新的列表，而大于等于基准元素的数则放入另外一个新的列表，然后将这两个新列表的快速排序的结果合并起来。例如：

```
def quicksort(arr):
    if len(arr) <= 1:                          #如果输入序列长度小于等于 1，则直接返回
        return arr
    pivot = arr[len(arr) // 2]                 #任意选择一个元素作为基准元素
                                               #将原序列一分为二
    left = [x for x in arr if x < pivot]       #left 是小于基准元素组成的子序列
    middle = [x for x in arr if x == pivot]    #middle 是等于基准元素组成的子序列
    right = [x for x in arr if x > pivot]      #right 是大于基准元素组成的子序列
    return quicksort(left) + middle + quicksort(right)  #对 left、right 重复上述过程

print(quicksort([49,38,27,97,76,13,27,49]))
```

输出：

```
[13, 27, 27, 38, 49, 49, 76, 97]
```

因为这个算法需要移动函数，所以这个算法的效率比较低，并且还要消耗更多的空间。

3.4.5 实战：迷宫问题

给定一个迷宫，指明迷宫的起点和终点，找出从起点出发到终点的有效路径，这就是迷宫问题

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

(maze problem)，如图 3-5 所示。

迷宫可以用二维数组来存储表示。例如，0 表示通路，1 表示障碍，2 表示终点。坐标以行和列表示，均从 0 开始。假设给定起点(0, 0)和终点(5, 5)，迷宫表示如下：

```
maze = [[0, 0, 0, 0, 0, 1],
        [1, 1, 0, 0, 0, 1],
        [0, 0, 0, 1, 0, 0],
        [0, 1, 1, 0, 0, 1],
        [0, 1, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 2]]
```

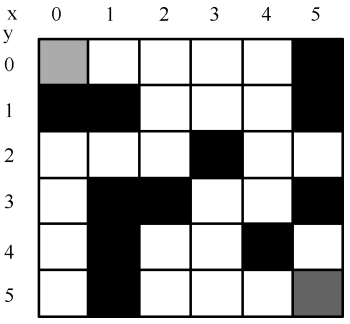


图 3-5　迷宫问题

迷宫求解问题可以描述为一个递归过程。

对于一个当前位置，判断该位置是否为终点(2)、墙(1)、已经走过(3)。如果不是上述情况，则说明该位置可通但未走过(0)。可从该位置走向其四邻(上、下、左、右四个位置)。对于每个邻接点，重复这个过程。例如：

```
def go_maze(x, y):
    #该位置是否为终点(2)、墙(1)、已经走过(3)
    if maze[x][y] == 2:
        print('到达终点：',x,",", y)
        return True
    elif maze[x][y] == 1:
        #print('墙：',x,",", y)
        return False
    elif maze[x][y] >= 3:
        #print('已经访问过：',x,",", y)
        return False

    #从该位置向四邻探索
    print('访问 %d,%d' % (x, y))
    maze[x][y] = 3              #标记该位置已经访问过

    # 向 4 邻探索
    if ((x < len(maze)-1 and go_maze(x+1, y))
        or (y > 0 and go_maze(x, y-1))
        or (x > 0 and go_maze(x-1, y))
        or (y < len(maze)-1 and go_maze(x, y+1))):

        return True

    maze[x][y] = 4     #此路也不通
    return False

go_maze(0, 0)
print(*maze,sep = '\n')
```

输出：

```
访问 0,0
访问 0,1
访问 0,2
访问 1,2
访问 2,2
访问 2,1
访问 2,0
访问 3,0
访问 4,0
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
访问 5,0
访问 1,3
访问 0,3
访问 0,4
访问 1,4
访问 2,4
访问 3,4
访问 3,3
访问 4,3
访问 5,3
访问 5,2
访问 4,2
访问 5,4
到达终点： 5 , 5
[[3, 3, 3, 3, 3, 1],
 [1, 1, 3, 3, 3, 1],
 [4, 4, 4, 1, 3, 0],
 [4, 1, 1, 3, 3, 1],
 [4, 1, 4, 3, 1, 0],
 [4, 1, 4, 3, 3, 2]]
```


思考：打印的位置还包括回退的位置，如何打印一个没有回退的路径？

3.5 函数对象和 lambda 表达式

3.5.1 函数对象

1. 函数也是对象

在 Python 中，函数也是对象，即 function 类型的对象，因此，可以和其他对象，如 int、list 对象一样使用。

- 用一个变量引用一个函数。
- 将函数作为另外一个函数的参数。
- 从一个函数里返回另外一个函数。
- 将函数存储在各种数据结构，如 list、tuple、set 里。

函数作为一个对象，也具有对象的三个属性，id、type 和值。例如，对于下面的函数 square()，可以检查它的 id 和 type 属性：

```
def square(x):
   return x*x

print(id(square))
print(type(square))
```

输出：

```
1767284622744
<class 'function'>
```

既然函数也是一个对象，当然可以将函数赋值给一个变量。例如：

```
fun = square
```

通过变量 fun 调用它引用的对象 square：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
fun(3.5)
```

输出：

```
12.25
```

函数名 square 和变量名 fun 都是同一个函数对象的名字：

```
print(id(fun))
print(id(square))
print(fun is square)
```

输出：

```
1767284521296
1767284521296
True
```

还可以把该函数赋值给更多的变量，唯一变化的是该函数对象的引用次数不断地增加，本质上，这些变量最终指向的都是同一个函数对象。

既然函数是对象，那么它就和其他对象一样可以放在一个如 list 这样的容器或集合里，也可以作为函数参数或返回值。

2．函数可以放在容器内

例如，下面的代码将两个函数 square()和 cube()放在 funcs 指向的字典对象中，通过 for…in 可以循环遍历这个 funcs 中的每个字典元素，并调用字典元素的值指向的函数：

```
def square(x):
    return x*x

def cube(x):
    """Cube of x."""
    return x*x*x

funcs = {
    'sq': square,
    'cb': cube,
}

x = 2
print(square(x))
print(cube(x))

for func in funcs:
    print(func, funcs[func](x))
```

输出：

```
4
8

sq 4
cb 8
```

当然也可以将函数放在其他容器，如一个 list 对象中：

```
fun_list= [square,cube]
for fun in fun_list:
    print(fun(x))
```

输出：

```
4
8
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

3．函数可以作为返回值

函数可以作为返回值。例如，下面的代码在函数 Squ()中将另外一个函数 square()作为返回值：

```
def Squ():
    return  square

f = Squ()
print(f(6))
```

输出：

```
36
```

4．函数可以嵌套

一个函数的内部可以定义另外一个函数。例如，下面的函数 Square()，函数内定义了另外一个函数 f()：

```
def Square(x):
   def f():
      return x*x
   y = f()     #调用函数 f()
   return y+x

print(Square(5))
```


调用 Square(5)时会执行其中的“y = f()”和“**return** y+x”，然后 print(Square(5))语句打印 Square(5)的返回结果：

```
30
```

注意

嵌套函数可以访问其包围环境中的数据。例如，函数 f()可以访问包围它的函数 Square()的参数(局部变量)x。

嵌套函数不能修改包围环境中的变量，除非该变量在嵌套函数中被声明为 nonlocal。例如：

```
def Square(x):
   def f():
      x = 2
      return x*x
   y = f()
   return y+x

print(Square(5))
```

输出：

```
9
```

函数 Square()中的 x 初始化值是指向 5 的变量，在函数 f()中的“x=2”实际上定义了一个函数 f()的局部变量，即指向这个对象 2，然后函数 f()返回这个 x 对象的乘积。也就是说，函数 f()返回的值是 4。而 y+x 中的 x 仍然是函数 Square()中的参数，其值仍然是调用函数 Square(5)时传递的值 5，而不是函数 f()中的局部变量 x =2。因此，最后的结果是 9。

但如果在函数 f()中声明了“**nonlocal** x”，则这个 x 将是函数 f()的包围环境，即函数 Square(x)中的 x。

当“x=2”时，即让 x 指向一个新的对象 2，因为这个 x 就是函数 Square()中的 x，当退出函数 f()后，函数 Square()中的 x 则指向这个对象 2，因此，最后的输出是 6。nonlocal 类似 global，函数中用 global 声明的变量是外部变量，nonlocal 声明的变量是其包围环境中的变量。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
def Square(x):
   def f():
      nonlocal x
      x = 2
      return x*x
   y = f()
   return y+x

print(Square(5))
```

输出：

```
6
```

5．函数可以作为其他函数的参数

函数可以作为其他函数的参数。例如，下面的代码中“SquList([2,3,4,5],square)”将一个函数 square()作为参数传递给函数 SquList()的形式参数 fun，而函数内部调用这个函数 fun()对 L 的每个元素 e 进行计算：

```
def SquList(L,fun):
    for e in L:
        print(fun(e),end=" ")

SquList([2,3,4,5],square)
```


输出：

```
4 9 16 25
```

3.5.2 lambda 表达式

1．lambda 表达式(匿名函数)

Python 中的函数一般都用 def 定义并有一个函数名，而 **lambda 表达式**(也称 **lambda 函数**或**匿名函数**)，是一个不用关键字 def 定义的没有函数名的函数，它主要用于定义简单的单行函数，即代码可以写在一行里，并且和普通函数一样，可以有参数列表。 其定义格式为：

```
lambda 参数: 语句
```

例如，下面定义一个带有参数 x、y 的 lambda 函数：

```
lambda x, y: x + y
```

输出：

```
<function __main__.<lambda>>
```

但这个 lambda 函数没有名字，所以无法使用它。既然函数是对象，当然也可以给 lambda 函数命名一个名字。例如，通过赋值运算符给 lambda 函数命名一个名字 add：

```
add = lambda x, y: x + y
```

可以调用 add 引用的这个 lambda 函数，并传递实际参数给这个 lambda 函数。

```
print(add(3, 5))
```

输出：

```
8
```

尽管只有一行代码，但 lambda 表达式是一个函数而不是普通的表达式，它可以接收传入的参数，而普通表达式则无法接收参数。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2．lambda 函数功能

lambda 函数主要用作函数的参数。有些函数需要接收另外一个函数作为其参数，使用 lambda 函数可以避免专门为这些函数写一个作为其参数的普通函数。

例如，对一个可迭代对象排序的内置函数 sorted()，其语法格式是：

```
sorted(iterable,key=None, reverse=False)
```

除要排序的可迭代对象形参 iterable 外，还有两个默认形参，其中的形参 key 必须是一个函数，该函数用于计算迭代访问的每个元素的 key(关键字)值。可选参数 reverse 表示按 key 大小逆序排序，默认不是逆序，即 reverse = False。例如：

```
alist = [-5,3,1,-7,9]
print(sorted(alist))
print(sorted(alist,reverse= True))
```

输出：

```
[-7, -5, 1, 3, 9]
[9, 3, 1, -5, -7]
```

上面的正序和逆序都是按照数据元素的值的大小进行比较而排序的。如果希望按照另外的某种大小比较方式(如绝对值)排序，就需要传递一个作为 key 参数的函数，这个函数用于计算一个数据元素的关键字值，函数 sorted()将按照这个关键字值进行排序。为此，需要写一个函数 Key()从数据元素 e 中得到其键值：

```
def Key(e):
  return abs(e)
```

可以将这个函数 Key()传给函数 sorted()的 key 形参：

```
print(sorted(alist,key=Key))
```

输出按照绝对值大小比较的有序列表：

```
[1, 3, -5, -7, 9]
```

上面的函数 Key()只有一行代码，完全可以用 lambda 函数代替这个函数 Key()作为函数 sorted()的 key 参数：

```
print(sorted(alist,key=lambda x: abs(x)))
[1, 3, -5, -7, 9]
```

这种用 lambda 函数代替普通函数作为其他函数形参的方式，避免了单独编写一个普通函数，使得代码更加清晰可读。

list 对象也有一个类似的排序方法 sort：

```
list.sort([key=..., reverse=...])
```

可以同样用 lambda 函数作为其 key 参数：

```
alist = [(2, 2), (3, 4), (4, 1), (1, 3)]
alist.sort(key = lambda e:e[1])
print(alist)
```

输出：

```
[(4, 1), (2, 2), (1, 3), (3, 4)]
```

这里的 lambda 函数“lambda e:e[1]”将一个数据元素 e 作为参数，e 是一个两个元素的 tuple，e[1]返回这个 tuple 对象的第二个元素的值，作为整个 lambda 函数的返回值。

下面再以一些常用的，可以接收函数参数的函数作为例子，进一步演示 lambda 表达式的定义与使用。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

3．内置函数 map()和内置函数 filter()

(1) 内置函数 map()。

内置函数 map()的规范是：

```
map(function, *iterable)
```

其中，iterable 是可变形参，即可以接收多个可迭代对象。内置函数 map()将第一个参数 function 指向的函数对象作用于每个可迭代对象的每个数据元素上。内置函数 map()返回一个迭代器对象(如 list、tuple 对象都是迭代器对象)，也可以用返回的迭代器对象构造一个 list 或 tuple 对象。例如：

```
def square(x):
    return x*x

ret = map(square,[3,4,5,6,7])      #将 square 作用于 list 对象的每个元素上
print(tuple(ret))                  #用返回的迭代器对象 ret 构造一个 tuple 对象
ret = map(square,[3,4,5,6,7])
print(list(ret))                   #用返回的迭代器对象 ret 构造一个 list 对象
```

输出：

```
(9, 16, 25, 36, 49)
[9, 16, 25, 36, 49]
```

用返回的迭代器对象构造一个 list 对象，即语句“list(map(function, iterable))”实际上类似执行下面的代码：

```
[function(x) for x in iterable]
```

下面的代码可以验证这一点：

```
#用返回的迭代器对象构造一个 list 对象
alist = list(map(square,[3,4,5,6,7]))
#类似执行下面的代码
blist = [square(x) for x in [3,4,5,6,7]]
print(alist)
print(blist)
```

输出：

```
[9, 16, 25, 36, 49]
[9, 16, 25, 36, 49]
```

实际上，内置函数 map()的第二个可变形参 iterable 可以接收多个可迭代对象，内置函数 map()用第一个函数参数 function 同时作用于这些可迭代对象的对应元素上，如果这些可迭代对象的数据元素个数不一样，则内置函数 map()只执行最小数目(数目最少的迭代器的数据元素个数)的操作。例如：

```
ret = map(lambda x , y : x * y, [1,4,3],[3,5,2,6])
print(tuple(ret))
```

以上代码传递了两个可迭代对象[1,4,3]和[3,5,2,6]，但前者只有三个元素而后者有四个元素，因此，只对三对对应元素((1,3)、(4,5)、(3,2))执行 lambda 表达式。

输出：

```
(3, 20, 6)
```

上述的 lambda 表达式返回的对象是一个数，也可以让 lambda 表达式返回一个其他形式的对象。例如，让 lambda 表达式返回一个 tuple：

```
ret = map(lambda x , y : (x * y,x+y), [1,4,3],[3,5,2,6])
print(tuple(ret))
```

输出：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
((3, 4), (20, 9), (6, 5))
```

内置函数 map()接收了两个 list 对象给可变形参 iterable。因为两个 list 对象的数据元素个数分别是三个元素和四个元素，所以，内置函数 map()用传递参数 function 的 lambda 表达式对它们的前三对数据元素执行计算。

再如：

```
ret = map(lambda x , y,z : (x * y*z,x+y+z), [1,4,3],[3,5,2,6],[7,8,9,10])
print(tuple(ret))
```

输出：

```
((21, 11), (160, 17), (54, 14))
```

(2) 内置函数 filter()。

内置函数 filter()的规范是：

```
filter(function or None, iterable)
```

它接收一个函数(或空值对象)和一个可迭代对象，返回的是一个新的迭代器对象，新的迭代器对象的每个元素都是被内置函数 function()判断为 True 的原迭代器中的元素。同样，可以用返回的迭代器对象构造一个 list 或 tuple 对象。例如：

```
numbers = range(-5, 5)
ret = filter(lambda x: x < 0, numbers)
less_than_zero = tuple(ret)
print(less_than_zero)
```

输出：

```
(-5, -4, -3, -2, -1)
```

上述代码中的内置函数 filter()接收一个 list 对象，并用一个 lambda 函数判断可迭代对象 numbers 的每个数是否为负数，内置函数 filter()返回的是这些负数构成的迭代器对象 ret，用这个 ret 传递给函数 tuple()以构造一个 tuple 对象，最后输出这个 tuple 对象。

总结

- 函数是 function 类型的对象，和其他对象一样，函数对象既可以作为函数参数、返回值，也可以存储在数据结构里。
- lambda 函数是一个匿名函数，主要用于单行代码的函数，经常用作其他函数的参数。
- 以函数作为参数的一些有用的内置函数如 map()、filter()。

3.6 模块和包

3.6.1 模块

1. 模块

函数是可以重复调用的程序块。在程序中使用他人已经编写好的函数代码，或者使自己编写的函数能被其他程序使用的方法是：**导入(import)**该函数所在的**模块(mudule)**。

Python 模块(mudule)就是包含 Python 语句的、文件名后缀(文件扩展名)是.py 的文件。这个文件中可以包含变量、函数和类等定义。

例如，创建一个叫作 hi 的模块，也就是创建一个 hi.py 文件：

```
def hello(name):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    print("hello")
    print(name)
```

模块 hi 里包含了一个叫作 hello()的函数。

2．导入(import)模块

在程序中如果需要使用其他模块中的函数等功能，就需要用关键字 import 导入相应的模块。导入模块的一般格式是：

```
import 模块名
```

要使用上面的模块 hi 中的 hello()函数，首先要 import(导入)hi 模块，格式如下：

```
import hi
```

注意

导入的模块只要说明模块名即可，不能有文件扩展名.py。程序代码中如果要使用模块中的对象，如函数，则需要用.运算符，即使用“模块名.函数名”访问具体的函数。例如，使用 hi.hello 访问模块 hi 中的函数 hello()：

```
import hi
hi.hello('小白')
```

输出：

```
hello
小白
```

Python 提供了许多标准模块，这些模块文件可以在 Python 安装目录的 lib 文件夹中找到。和导入自己编写的模块一样，可以导入那些别人编写好的模块。例如，导入一个用于数学计算的模块 math，其中有一个表示圆周率的变量 pi：

```
import math

print(math.pi)
print(math.sin(1.57))  #1.57 约等于 math.pi/2
print(math.sin(math.pi/2))
```

输出：

```
3.141592653589793
0.9999996829318346
1.0
```

如果忘记在函数名前添加“模块名.”，则会报错：“名字 xxx 未定义”。例如：

```
import math
print(sqrt(2))     #sqrt 是求平方根函数
```

输出：

```
---------------------------------------------------------------------------
NameError                 Traceback (most recent call last)
<ipython-input-8-692e4d48a502> in <module>()
      1 import math
----> 2 print(sqrt(2))     #sqrt 是求平方根函数

NameError: name 'sqrt' is not defined
```

3．重命名导入模块

可以用“import ...as”对一个导入进来的模块重新命名。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import math as m              #将导入的模块 math 命名为 m
print(m.pi)                   #通过 m 而不是 math 去引用其中的名字
```

输出：

```
3.141592653589793
```

此时，只能用重命名的 m.pi 而不能用 math.pi 去访问 math 中的 pi。

4．导入单独名字

可以用“from...import”从一个模块导入一个单独的名字，导入的这个名字就不用在前面添加“模块名.”前缀了。例如：

```
from math import sqrt     #从 math 模块导入名字 sqrt
print(sqrt(2))            #sqrt 是求平方根函数
```

输出：

```
1.4142135623730951
```

5．导入所有名字

还可以通过“**from**…**import** *”导入模块中的所有名字，该模块中的名字就可以直接使用，而不再需要模块名前缀了：

```
from math import *
print(pi)             #math 模块中的 pi 变量
print(sin(1.57))      #math 模块中的 sin 函数
print(sqrt(2))        #math 模块中的 sqrt 函数
```

输出：

```
3.141592653589793
0.9999996829318346
1.4142135623730951
```

模块中的对象，如常量、函数、类等，可被不同的程序代码重复使用，即模块使得代码可以“**复用**”(如重复多次调用一个模块中的函数)。例如，前面多次使用的函数 print()、math 模块中的函数 sqrt()等。

模块还有一个好处就是可以避免“**名字冲突**”，因为程序代码中不可避免地会使用他人的代码，如果不同的人使用了同一个名字就会引起冲突，但如果这些名字属于不同的模块，就可以通过模块名来区分它们。

因此，为了避免名字冲突，应该尽量避免“from...import”和“from ... import *”这种将名字直接导入的方法，而推荐使用“import ...”导入模块的方法。

前面介绍过 Python 的内置函数 **type()**可以用于查询一个对象的类型，此外，还有两个函数 **dir()**和 **help()**，可分别用于显示一个对象的所有属性和查询一个对象(如函数)的文档字符串。通过这三个“内省函数”可以获得一个对象的帮助信息。

6．函数 dir()

内置函数 dir()返回一个对象的所有属性的名字。其函数格式是：

```
dir([object])
```

如果没有指定可选的参数对象 object，则返回当前作用域中的所有名字，否则就返回指定的参数对象的 object 的所有属性。例如：

```
>>>dir()
```

返回当前作用域中的所有名字：

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

假如继续执行：

```
>>>alist = [2,5,8]
>>>dir()
```

则可以看到返回的名字的列表里多了一个新的名字 alist：

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'alist']
```

假如再执行：

```
>>>b = 3.14
>>>dir()
```

则可以看到返回的名字的列表里又多了一个新的名字 b：

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'alist', 'b']
```

假如在调用函数 dir()时输入一个对象，则函数 dir()返回这个对象的所有属性。例如：

```
>>>dir(alist)
```

将显示 list 类型对象 alist 的所有属性：

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__','__new__','__reduce__','__reduce_ex__','__repr__','__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__sub-
classhook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

假如再执行：

```
>>>dir(b)
```

将显示 float 类型对象 b 的所有属性：

```
['__abs__', '__add__', '__bool__', '__class__', '__delattr__', '__dir__',
'__divmod__', '__doc__', '__eq__', '__float__', '__floordiv__', '__format__',
'__ge__', '__getattribute__', '__getformat__', '__getnewargs__', ...]
```

如果输入一个模块名，则显示这个模块中的所有属性，如模块中的函数名、类名、变量名等。例如：

```
>>>import sys
>>>dir(sys)
```

将显示 sys 模块的所有名字(属性)：

```
['__displayhook__', '__doc__', '__excepthook__', '__interactivehook__',
'__loader__','__name__','__package__','__spec__','__stderr__','__stdin__',
'__stdout__', '_clear_type_cache', '_current_frames',
...]
```

如果输入自定义的模块名 hi：

```
>>>import hi
>>>dir(hi)
```

则显示 hi 模块的所有属性：

```
['__builtins__','__cached__','__doc__','__file__','__loader__','__name__',
 '__package__', '__spec__', 'hello']
```

7．函数 help()

函数 help()打印一个对象的帮助信息。例如，通过函数 help()了解函数 math.sin()的函数说明：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import math
help(math.sin)
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
```

如果通过一个对象的__doc__属性直接打印一个对象的文档串，则显示类似的帮助文档。例如：

```
print(math.sin.__doc__)
```

输出：

```
sin(x)
Return the sine of x (measured in radians).
```

读者还可完成以下练习。

执行如下的 help 和 dir 命令，查询这些模块或函数的帮助文档或属性。

help(len)——内置函数 len()的帮助文档，注意参数是 len 而不是 len()。

help(sys)——系统模块 sys（必须先导入 sys)的帮助文档。

dir(sys)——系统模块 sys 里定义的符号(属性)。

help(sys.exit)——sys 模块里的函数 exit()的帮助文档。

help('xyz'.split)——字符串的 split()方法的帮助文档，可以通过 str 类型或 str 类型的对象，如 xyz 调用这个方法 help('xyz'.split)，等价于 help(str.split)。

help(list)——list 对象的帮助文档。

dir(list)——list 对象的所有属性，包括对象的所有方法和属性。

help(list.append)——list 对象的 append()方法。

通过这些帮助信息，就可以了解如何使用某个函数或对象的方法或属性。例如，使用 list 的 append()方法可以向一个 list 对象的最后添加一个元素：

```
alist=[1,2,3]
alist.append('hello')
print(alist)
```

输出：

```
[1, 2, 3, 'hello']
```

8．模块的__name__属性

每个模块都有一个__name__属性，当这个模块被其他程序(模块)通过 import 方式导入时，其值就是这个模块的模块名。例如：

```
>>> print(hi.__name__)
```

输出：

```
hi
```

但如果一个模块作为主程序执行，即在控制台窗口通过 Python 解释器运行这个脚本时，这个模块的属性__name__就是 main，即说明这个模块是主程序。例如，编写如下的脚本文件 hi.py：

```
def hello(name):
    print("hello")
    print(name)

print(__name__)   #打印该模块自身的__name__属性
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

在控制台窗口运行这个脚本：

```
python hi.py
```

输出结果如下：

```
__main__
```

因此，在编写脚本文件时，经常在程序中根据__name__==__main__判断脚本是否作为主程序运行从而执行特定的程序代码。例如，修改上述的 hi.py 代码：

```
def hello(name):
    print("hello")
    print(name)

print(__name__)
if (__name__ == '__main__'):
    hello('Li ping')
else:
  print('hi.py 作为模块导入而不是作为主程序运行!')
```

该程序通过 if(__name__ ==' __main__')判断脚本是否在主程序执行，如果作为主程序执行，则执行 hello(‘Li Ping’)，否则就执行另外的 print()语句。例如，执行：

```
python hi.py
```

输出：

```
__main__
hello
Li Ping
```

3.6.2 sys 模块(Python 解释器接口)

sys 模块负责与 Python 解释器的交互，提供一系列的函数和变量，用于操控 Python 的运行时环境。

1. sys.argv

在 Python 解释器下执行一个脚本时，会通过 sys.argv 变量向这个脚本传递一个命令行参数列表。其中，第一个元素 sys.argv[0]是脚本程序的完整路径或文件名(取决于操作系统)。例如，编写一个叫作 abc.py 的脚本：

```
import sys
print('脚本名: ', sys.argv[0])
```

在 Python 解释器中执行这个脚本:

```
python abc.py
```

在 Windows 系统上将输出下面的信息:

```
脚本名: abc
```

如果执行这个脚本时，新提供了其他的参数，那么从第二个元素 sys.argv [1]起就是这些新提供的参数。例如，abc.py 脚本内容如下：

```
import sys
print('脚本名: ', sys.argv[0])
print('Hello  ', sys.argv[1])
```

在 Python 解释器中执行如下脚本：

```
python abc.py wang
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

程序代码中 sys.argv[0]是模块名 abc，而 sys.argv[1]是参数 wang，在 Windows 系统上将输出下面的信息：

```
脚本名：abc
Hello  wang
```

2．sys.path（模块搜索路径）

当 import 一个模块时，Python 会在一些位置查找是否存在相应名字的模块，首先查找是否在内在(built-in)模块中，如未找到就在当前目录查找，如仍未找到，就在 sys.path 变量指定的目录里查找。

sys.path 是一个字符串列表，用于指定模块的搜索路径，包括环境变量 PYTHONPATH 里的目录和安装目标的默认值。Python 解释器在导入模块时，会在这些路径中查找相应的模块。

例如，可输出 sys.path 中的目录内容：

```
import sys
print(sys.path)
```

输出：

```
['', 'E:\\Anaconda\\python36.zip', 'E:\\Anaconda\\DLLs', 'E:\\Anaconda\\lib',
'E:\\Anaconda', 'E:\\Anaconda\\lib\\site-packages', 'E:\\Anaconda\\lib\\
site-packages\\win32', 'E:\\Anaconda\\lib\\site-packages\\win32\\lib',
'E:\\Anaconda\\lib\\site-packages\\Pythonwin', 'E:\\Anaconda\\lib\\
site-packages\\IPython\\extensions', 'C:\\Users\\s\\.ipython']
```


sys.path 是一个字符串的 list，其中，每个元素表示一个文件路径，也可以在自己编写的程序中将特定的文件路径通过 list 的 append()添加到这个 list 中。

```
sys.path.append("python/my_code")
print(sys.path)
```

输出：

```
['', 'E:\\Anaconda\\python36.zip', 'E:\\Anaconda\\DLLs', 'E:\\Anaconda\\lib',
'E:\\Anaconda', 'E:\\Anaconda\\lib\\site-packages', 'E:\\Anaconda\\lib\\
site-packages\\ win32', 'E:\\Anaconda\\lib\\site-packages\\win32\\lib',
 'E:\\Anaconda\\lib\\site-packages\\ Pythonwin', 'E:\\Anaconda\\lib\\
site-packages\\IPython\\extensions', 'C:\\Users\\s\\ .ipython', 'python/my_code']
```

3．sys.exit()（退出函数）

退出函数 sys.exit()用于退出 Python 脚本程序，该函数可以带一个整数作为参数，用于表示程序退出的状态(不同的整数表示不同的退出情形)。不同操作系统用不同整数的表示程序退出状态。通常，传递整数 0 表示程序正常退出。当调用函数 sys.exit()时，它将引发 SystemExit 异常，该异常允许清理函数在异常处理的 try / except 模块的 finally 子句中起作用。例如：

```
import sys
sys.exit(0)
```

抛出错误异常：

```
An exception has occurred, use %tb to see the full traceback.
SystemExit: 0
```

如果要退出解释器而不是单个脚本程序，则可以使用内置函数 exit()，该内置函数直接退出并关闭解释器：

```
exit(0)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

4．sys.executable

sys.executable 中保存 Python 解释器的完整路径。例如：

```
import sys
sys.executable
```

输出：

```
'C:\\Users\\s\\Anaconda3\\python.exe'
```

5．sys.platform

sys.platform 值为平台标识符。例如：

```
import sys
sys.platform
```

输出：

```
'win32'
```

通过检查 sys.platform 值，可根据不同平台导入不同的模块，或者根据不同平台执行不同的处理代码。例如：

```
os = sys.platform
if os == "win32":
    # 使用 Windows 平台相关的代码或模块
    pass
elif os.startswith('linux'):
    # 使用 Linux 平台相关的代码或模块
    import subprocess
    subprocess.Popen(["ls, -l"])
```

6．sys.getrefcount()

函数 sys.getrefcount() 返回一个对象的引用计数，该计数通常比用户预期的多一个，因为它包含作为函数 getrefcount() 形参的临时引用。例如：

```
import sys
a = 20489                  #a 引用的 20489 的引用计数为 1
sys.getrefcount(a)         #实参 a 传递给函数形参，使 20489 的引用计数为 2
```

输出：

```
2
```

继续执行下面的命令：

```
b = a

sys.getrefcount(b)          #b 引用了 a 引用的对象，20489 的引用计数变为 3
```

输出：

```
3
```

当 a = 20489 引用创建的字符串对象 20489 时，该对象的引用计数为 1；将 a 作为实参传递给 sys.getrefcount(obj) 时，形参 obj 又引用了 a 引用的对象，使得 20489 的引用计数变为 2；当 sys.getrefcount(b) 时，即 b 引用 a 引用的对象时，20489 的引用计数变为 3；当一个对象的引用计数变为 0 时，Python 才开始销毁这个对象，并回收这个对象占用的内容。

7．sys.getsizeof()

函数 sys.getsizeof() 返回一个对象占用的内存的字节数。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import sys
a = 25
sys.getsizeof(a)
```

输出：

```
28
```

即整数 a 占用了 28 字节。

8．sys.stdin、sys.stdout、sys.stderr

sys.stdin、sys.stdout、sys.stderr 分别映射到与解释器的标准输入、标准输出和错误流相对应的文件对象。除脚本外，sys.stdin 用于提供给解释器的所有输入，而 sys.stdout 用于函数 print()的输出。解释器自己的提示及其错误消息转到 sys.stderr。例如：

```
import sys
user_input = sys.stdin.readline()
```

程序等待用户输入，如果输入如下内容：

```
hello world
```

然后，执行下列语句：

```
print("Input : " + user_input)
```

则输出：

```
Input : hello world
```

9．sys.getdefaultencoding()

函数 sys.getdefaultencoding()获取系统当前编码，系统默认编码是 UTF-8。

```
sys.getdefaultencoding()
```

输出：

```
'utf-8'
```

10．sys.setdefaultencoding()

函数 sys.setdefaultencoding()设置系统默认编码。例如，执行 setdefaultencoding ('utf8')时，则将系统默认编码设置为 UTF-8。

3.6.3 伪随机数发生器模块

1．random 模块

random 模块为不同的分布提供伪随机数生成器。例如，为整数提供从一个范围里均匀地选取一个整数的函数；为序列提供均匀地选择一个元素、生成随机排列、用于随机抽样的函数；为实数提供计算均匀、正态(高斯)、对数正态、负指数、伽马和贝塔分布的函数。其中，为了生成角度分布，可以使用 von Mises 分布。

几乎所有模块函数都依赖于基本函数 random()，它在半开放范围[0.0,1.0)内均匀生成随机浮点数。Python 使用 Mersenne Twister 作为核心生成器，它产生 53 位精度浮点数，周期为 $2^{19937}-1$，其底层的 C 实现线程既快又安全。 Mersenne Twister 是最广泛使用的随机数发生器之一，但因其具有完全确定性，所以它不适用于所有目的，且完全不适用于加密。

下面是 random 模块的一些常用随机数函数。

- random.**random**()　生成[0.0, 1.0)之间的浮点数。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- random.**uniform**(a, b)　如果 a<b，则生成[a,b]之间的浮点数，否则，生成[b,a]之间的浮点数。
- random.**choice**(sequence)　从序列 sequence 中获取一个随机元素。
- random.**randrange**([start], stop[, step])　在 range(start, stop, step)中随机选择一个元素，等价于 choice(range(start, stop, step))。

例如，random.randrange(10, 100, 2)，结果相当于从[10, 12, 14, 16, ..., 96, 98]序列中获取一个随机数。

- random.**randint**(a, b)　生成整数 a 和 b 之间的整数，且必须满足 a<=b，相当于 randrange (a, b+1)。
- random.**shuffle**(x[, random])　随机置换序列 x(将序列 x 中元素的位置随机置换/打乱顺序)，可选参数 random 是一个返回[0.0,1.0)之间浮点数的随机函数，默认就是 random.random()函数。
- random.**sample**(x, k)　从总体序列或集合 x 中随机选择 k 个元素，该函数可用于无替换随机抽样。函数 random.sample()不会修改原有序列，而只是返回一个 list 对象。

下面是一个简单的使用示例。

```
import random
a = random.random()            #[0.0, 1.0)之间的随机浮点数(除 1 外)
b = random.uniform(100, 1)     #[1.0, 100.0)之间的浮点数，可能不包括 100.0
c= random.randint(-10, 80)     #随机整数
print(a,'\t',b,'\t',c)
r = random.choice(r'dfs*d=!kh#^h@')    #在字符串 r'dfs*d=!kh#^h@'中随机选择一个
print(r)
p = ["Python", "C", "小白", "a.hwdong.com"]
print(random.choice(p))        #从列表 x 中任意选择 1 个数
random.shuffle(p)              #重排序：打乱列表 x 中元素的顺序
print(p)
alist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
a = random.sample(alist, 5)    #从 list 中随机获取 5 个元素，返回 1 个 list 对象
print(a)
```

输出：

```
0.3492832859352474    77.17952854183415    66
s
小白
['Python', 'C', '小白', 'a.hwdong.com']
[10, 8, 5, 3, 9]
```

2．种子(seeding)

每次调用 random()等随机数生成函数时都会产生不同的值，这对于生成不断变化的随机数是有用的，但有时则希望以不同方式处理相同的数据集。例如，对于两个不同的算法，从数据集随机取出相同的一组数据，然后比较不同算法对这组数据处理的差别。这时可以用函数 seed()初始化随机数发生器，以便让它产生预期的一组值。例如，两次运行下面的程序，都会产生一样的一组随机数：

```
import random
def f():
    random.seed(1)
    for i in range(5):
        print('{:04.3f}'.format(random.random()), end=' ')
    print()

    for i in range(3):
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            print(random.randint(-5, 5), end=' ')
        print()
    f()
    f()
```

输出：

```
0.134 0.847 0.764 0.255 0.495
2 2 5
0.134 0.847 0.764 0.255 0.495
2 2 5
```

3.6.4 包

1. 包(package)

一个大的软件项目会包含很多模块文件，如果将这些模块文件都放在一个文件夹中，则难以管理并跟踪它们。如同计算机中的资源管理器使用不同文件夹分类管理文件一样，这些模块文件也可以放在不同文件夹(目录)中。

Python 以包的形式将相关模块文件组合在不同文件夹中。Python 的包的概念很简单，可以理解为，包就是一个文件夹，其中包含了一个__init__.py 文件。这个__init__.py 文件可以是一个空文件，也可以包含一些 Python 语句。也就是说，包就是一个包含__init__.py 文件的文件夹，包名就是文件夹名。在一个包(文件夹)中可能还包含其他的包(文件夹)或模块文件，这些包中又会包含其他的包和模块文件，从而形成一个层次性的目录结构。

创建包很简单，就是利用操作系统自带的层次性文件系统创建包对应的文件夹。例如，创建一个叫作 myProj 的文件夹，在该文件夹下创建一个空的__init__.py 和两个模块文件 hello.py、hi.py。这个叫作 myProj 的文件夹就是一个包，包名就是文件夹名 myProj，文件夹和其中的文件形成如下的目录结构。

```
myProj
 |_ __init__.py
 |_ hello.py
 |_ hi.py
```

例如，hello.py 的内容如下：

```
def hello(name):
  print('hello,',name)

def f():
  print('你好')
```

例如，hi.py 的内容如下：

```
def welcome(city):
   print('hi,welcome to ',city)
```

例如，这个文件夹 myProj 的路径已经在 sys.path 的某个路径中(如果该文件夹路径不在 sys.path 中，则可通过 sys.path.a ppend()将其添加到这个对象中)，在程序中就可以导入这个包中的模块文件，并调用其中的对象(如函数)：

```
import myProj.hello, myProj.hi
myProj.hello.hello('Li ping')
myProj.hi.welcome('ShangHai')
```

输出：

```
hello, Li ping
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
hi,welcome to  ShangHai
```

当然，也可以采用 from...import 导入单个模块或名字。

例如，导入单个模块：

```
from myProj import hello
hello.hello('Li ping')
```

输出：

```
hello, Li ping
```

可以导入单个模块并重命名。例如：

```
from myProj import hello  as he
he.hello('Li ping')
```

输出：

```
hello, Li ping
```

也可以导入模块中的具体对象(如函数)。例如：

```
from myProj.hello import hello  as heo
heo('Li ping')
```

输出：

```
hello, Li ping
```


当然，还可以导入整个包。例如：

```
import myProj
myProj
```

尽管语法上没有问题，但是这种方式并没有将该包中的模块名导入当前的名字空间中，因此，不能访问该包中的模块：

```
myProj.hello.hello('Li ping')
```

这种导入包的方式没有用处。

2．包的初始化

如果程序包目录中存在名为__init__.py 的文件，则在导入程序包或程序包中的模块时会执行该文件。该功能可以用于执行包初始化代码，如包级数据的初始化。

例如，__init__.py 的代码如下：

```
print('调用__init__.py for {__name__}')
NAMES = ['Li Ping', 'Wang Hao', 'Zhang Ping']
```

当导入包或包中的模块时，会首先执行__init__.py 中的代码，如上面的输出语句和初始化一个全局变量 NAMES：

```
import myProj
```

输出：

```
调用__init__.py for myProj
```

执行：

```
myProj.NAMES
```

输出：

```
['Li Ping', 'Wang Hao', 'Zhang Ping']
```

模块中的代码可以访问用 import 导入的包中的全局变量，如 myProj.NAMES。例如，修改 hello.py 模块文件，导入 myProj 的 NAMES 全局变量：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
def hello(name):
  print('hello,',name)

def f():
  from myProj import NAMES
  print(NAMES[0])
```

执行：

```
from myProj import hello
hello.f()
```

输出：

```
调用__init__.py for myProj
Li Ping
```

__init__.py 也可用于实现从包中自动导入模块。import myProj 仅将名称 myProj 放在调用程序的本地符号表中，而不导入任何模块。但是，如果 myProj 目录中的__init__.py 包含以下内容：

```
print(f'调用__init__.py for {__name__}')
import myProj.hello, myProj.hi
NAMES = ['Li Ping', 'Wang Hao', 'Zhang Ping']
```


则当执行 import myProj 时，包中的模块 hello 和 hi 就被自动导入了。下面的语句就能正常工作了：

```
import myProj
myProj.hello.hello('Li ping')
```

对于模块，可以通过 import *将模块中的所有对象导入到本地符号表中。例如：

```
>>>dir()
```

显示本地符号表：

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
```

继续执行：

```
>>>from myProj.hello import *
>>>dir()
```

显示本地符号表：

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'f', 'hello']
```

可以看到，myProj.hello 模块中的对象，如函数 hello()和 f()，都已导入本地符号表中。

对于包也可使用 import *。例如：

```
from myProj import *
```

显示结果：

```
['NAMES', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__']
```

可以看到，已导入包 myProj 中的全局变量 NAMES，但并没有导入包及其包含的模块中的所有对象。

3. __all__变量

Python 规定，如果在__init__.py 中定义了一个**__all__**变量，且该变量中包含一些模块名，那么使用 import *，如 from myProj import *，就会导入**__all__**变量中的模块中的所有对象。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

例如，修改，__init__.py 文件：

```
print('调用__init__.py for {__name__}')
NAMES = ['Li Ping', 'Wang Hao', 'Zhang Ping']
__all__ = ['hello','hi']
```

可以看到，此时本地符号中就包含了**__all__**变量中的模块名：

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'hello', 'hi']
```

可以使用这些模块中的对象，如：

```
>>>hello.f()
```

输出：

```
Li Ping
```

同样，调用 hi 模块的函数 welcome()：

```
>>>hi.welcome('Bei Jing')
```

输出：

```
hi,welcome to  Bei Jing
```

在__init__.py 中定义的**__all__**变量是针对整个包的，也可以在单独的模块文件中定义**__all__**变量，以说明该模块文件中的哪些对象可以被其他程序 import。例如，修改 hello.py 文件：


```
__all__ = ['hello']

def hello(name):
  print('hello,',name)

def f():
  from myProj import NAMES
  print(NAMES[0])
```

说明当该模块被其他程序导入时，只有函数 hello()是可见的。

```
>>>dir()
```

输出：

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
```

执行：

```
from myProj.hello import *
dir()
```

输出：

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'hello']
```

因此，可以执行：

```
hello('李平')
```

输出：

```
hello, 李平
```

但不能执行：

```
f()
```

会产生错误：

```
Traceback (most recent call last):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
  File "<pyshell#5>", line 1, in <module>
    f()
NameError: name 'f' is not defined
```

__all__总结

- 如果没有为一个包定义__all__，则当使用 import *时不会导入任何对象。
- 如果没有为一个模块定义__all__，则当使用 import *时将导入模块中的所有对象，否则仅导入__all__中定义的对象。

4．子包

包中可以嵌套子包(如同文件夹可以包含子文件夹一样)，子包中还可以嵌套子包的子包。包之间可以一直这样嵌套下去(正如文件夹可以一直嵌套一样)。例如：

```
myProj
 |_ __init__.py
 |_ util
     |_ __init__.py
     |_ img_process.py
     |_ math_util.py
 |_ game
     |_ __init__.py
     |_ init_game.py
     |_ scene
        |_ __init__.py
        |_ init_scene.py
        |_ render.py
        |_ update.py
 |_ hello.py
 |_ hi.py
```

在 myProj 包中除三个模块文件__init__.py、hello.py 和 hi.py 外，还有两个子包 util 和 game，在每个子包下又有一些模块文件或包，如 game 包下有两个模块文件__init__.py、init_game.py 和一个包 scene，包 scene 下又有一些模块文件。

无论这种嵌套的包的层次结构多么复杂，都可以使用 import 语句，唯一需要注意的是，需要用额外的 . 表示完整的正确嵌套包含关系。例如：

导入单独的模块：

```
import myProj.util.math_util
```

导入单独的模块名：

```
from myProj.util import math_util
```

假设 render 模块有一个函数 render()，则也可以直接导入单独的函数：

```
from myProj.game.scene.render import render
```

3.6.5 Matplotlib 包

Python 提供了一个专门用于绘图的工具包 Matplotlib。Windows 系统下以管理员权限打开命令行窗口，然后执行下列命令进行安装：

```
pip install matplotlib
```

Linux 或 Mac 可以管理员权限执行下列命令进行安装：

```
sudo pip install matplotlib
```

matplotlib 的 pyplot 模块提供了简单的绘图函数，可以用下面的代码导入 matplotlib.pyplot 模块并命名为 plt，以避免在代码中写入一长串的 matplotlib.pyplot。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import matplotlib.pyplot as plt
```

jupyter 环境中可以用命令“%matplotlib inline”将图形嵌入在浏览器网页中。

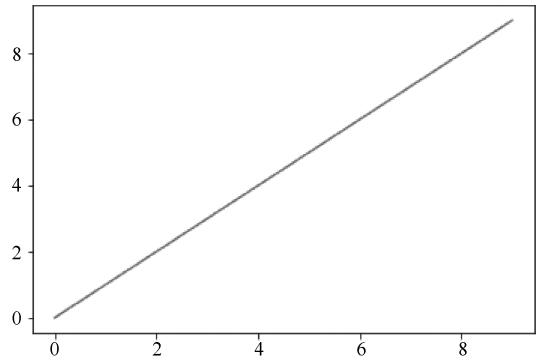
```
%matplotlib inline
```

pyplot 模块的函数 plot()可以直接绘制 2D 数据。例如：

```
y = [i for i in range(10)]
print(y)
plt.plot(y)          # 绘制 y 作为纵轴坐标点构成的图形
plt.show()           # 调用 plt.show()显示图形
```

输出并显示绘制图形：

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

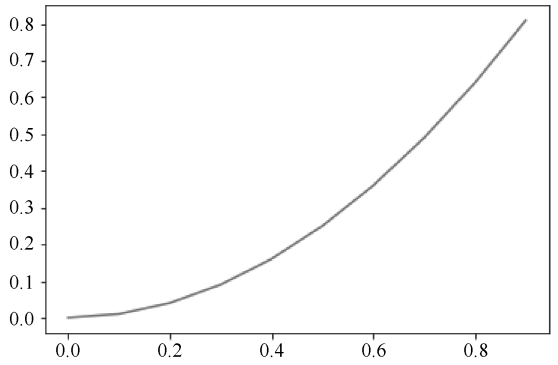


尽管只给了纵轴坐标的数组 y，函数 plot()默认自动生成从 0 开始的横轴坐标。当然，可以分别传递 2 个数组表示 x 和 y 坐标。例如：

```
x = [i*0.1 for i in range(10)]
y = [xi**2 for xi in x]
print(["{0:0.2f}".format(i) for i in x])
print(["{0:0.2f}".format(i) for i in y])
plt.plot(x, y)          #绘制(x,y)坐标点构成的图形
plt.show()              #调用 plt.show()显示图形
```

输出：

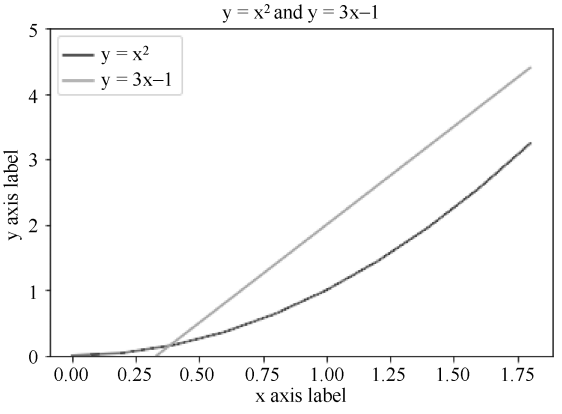
```
['0.00', '0.10', '0.20', '0.30', '0.40', '0.50', '0.60', '0.70', '0.80', '0.90']
['0.00', '0.01', '0.04', '0.09', '0.16', '0.25', '0.36', '0.49', '0.64', '0.81']
```



可以绘制几个曲线。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

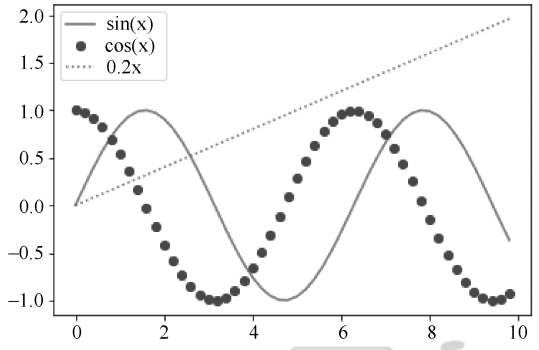
```
x = [i*0.2 for i in range(10)]
y = [xi**2 for xi in x]
y2 = [3*xi-1 for xi in x]

plt.plot(x, y)              #绘制(x,y)坐标点构成的图形
plt.plot(x, y2)
plt.ylim(0,5)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('y=$x^2$ and y=3x-1')
plt.legend(['y=$x^2$', 'y=3x-1'])
plt.show()                  #调用 plt.show()显示图形
```



其中，pyplot 模块的函数 title()用于给图起一个标题，而函数 legend()则给每个绘制的曲线起一个名字，函数 xlim()和 ylim()分别用于限定 x 和 y 坐标的范围，函数 xlabel()和 ylabel()分别用于给 x 轴和 y 轴分配一个标签。可以看到不同的图形将自动用不同的颜色显示。

函数 plot()还可以接收一些参数，用于定制绘制的图形的样式。例如：

```
import math
x = [i*0.2 for i in range(50)]
y = [math.sin(xi) for xi in x]
y2 = [math.cos(xi) for xi in x]
y3 = [0.2*xi for xi in x]
plt.plot(x, y,'r-')
plt.plot(x, y2,'bo')
plt.plot(x, y3,'g:')
plt.legend(['sin(x)', 'cos(x)','0.2x'])
plt.show()
```

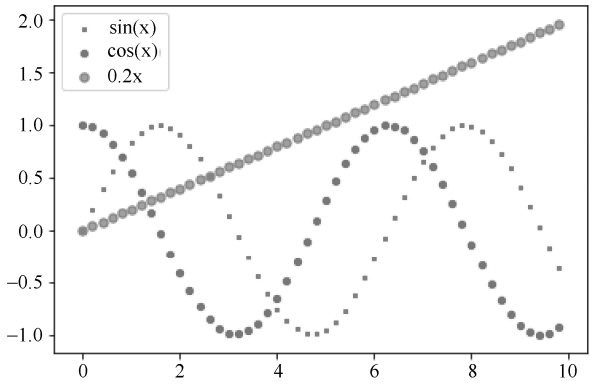


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

其中，'r-'的 r 表示红色（red）；-表示短线；'bo'的 b 表示蓝色（blue）、o 表示以圆点；'g:'的 g 表示绿色（green）；:表示虚线。

pyplot 模块除函数 plot()可以绘图外，还有其他的一些函数也可用于绘制其他类型的图，如 函数 scatter()用于绘制散乱点图。例如：

```
import math
x = [i*0.2 for i in range(50)]
y = [math.sin(xi) for xi in x]
y2 = [math.cos(xi) for xi in x]
y3 = [0.2*xi for xi in x]
plt.plot(x, y,'r-')
plt.plot(x, y2,'bo')
plt.plot(x, y3,'g:')
plt.legend(['sin(x)', 'cos(x)','0.2x'])
plt.show()
```



总结

- 后缀是**.py** 的 Python 程序文件称为**模块**。可以用关键字 import 导入一个模块(如 xxx)到程序中，模块 xxx 中的名字如 name，可以通过 xxx.name 访问。import xxx as yy 用于在导入模块 xxx 时给模块 xxx 起一个别名 yy，访问模块中的名字就要用这个别名作为前缀。
- from ... import 可以导入模块中的一个名字(如 from xxx import name 导入模块 xxx 的单个名字 name)，也可以导入模块中的所有名字(如 from xxx import *导入模块中的所有名字)。
- 包就是一个包含__init__.py 文件的文件夹。这个__init__.py 可以是空的文件，也可以包含一些 Python 命令(如执行一些初始化)。包将所有模块文件组织成一个层次结构。
- sys 模块是 Python 解释器交互的接口，如向脚本程序传递命令行参数，添加工作路径等。
- random 随机数模块可以用于生成各种随机数，如从一个序列对象里随机选择元素，或者得到一个序列的随机排列等。
- Python 提供了一个专门用于绘图的工具包 Matplotlib。

3.7 实战：Pong 游戏

3.7.1 Pygame 游戏库介绍

Pygame 实际上是 C 语言编写的游戏库 SDL(Simple Direct Layer))的 Python 的包裹封装，SDL 是一个跨平台的多媒体库(也被用来开发游戏)。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

要使用 Pygame，首先要在命令行用 pip 安装 Pygame 和其他模块(库)：

```
pip install pygame
```

若要在程序中使用 Pygame，则必须 import。例如：

```
import pygame
```

3.7.2 用 Pygame 编写游戏

1. 游戏的骨架

无论使用何种编程语言和游戏开发库，任何游戏都必须遵循同样的框架。游戏一开始会进行一些初始化工作，然后显示开始画面，根据用户的输入和随着时间的变化，游戏中的元素(对象)会发生变化，从而导致画面产生变化。游戏不断地处理用户输入、更新游戏的数据、绘制场景，这个过程不断循环。因此，所有游戏都具有如下的程序结构或框架。

```
初始化
无限循环:
    处理用户输入
    更新游戏状态
    绘制并显示场景
退出程序
```


2. Pygame 游戏的骨架

下面是一个最简单的 Pygame 游戏的骨架(在游戏画面中间显示一个圆)：

```
import sys
import pygame  #导入 pygame 模块

def game_engine():
    # 1. 初始化
    pygame.init()   #初始化 Pygame

    #设置窗口的模式，(680，480)表示窗口(宽度，高度)
    #此函数返回一个用于绘制的 surface 对象(相当于一块画布)
    surface = pygame.display.set_mode((680, 480))

    pygame.display.set_caption('Game Engine')   #设置窗口标题

    #2. 循环，直到游戏结束
    running = True
    while running == True:
        #2.1 处理(键盘、鼠标等)事件
        for event in pygame.event.get():          #返回当前的所有事件
                if event.type == pygame.QUIT:     #接收到窗口关闭事件
                    running = False               #退出游戏

        # 2.2 更新游戏的数据
        #...

        #2.3 在 surface 对象上绘制并显示
        #(红、绿、蓝)颜色 color 绘制一个给定圆心和半径的圆
        color = (255, 128, 5)
        pygame.draw.circle(surface, color, (320,240),20)
        pygame.display.flip()  #update()

    pygame.quit()       # 3.退出程序
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
if __name__ == "__main__":
    game_engine()
```

其中，pygame.event.get()返回当前的所有事件，pygame.quit()与 pygame.init()作用相反，它将注销 Pygame 库，程序应该在调用 sys.exit()前调用 pygame.quit()。

执行上面的程序，可以看到如图 3-6 所示的游戏画面中间的一个圆。

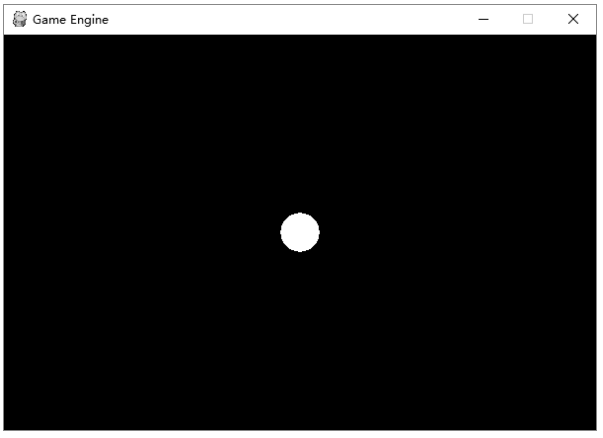


图 3-6　游戏画面中间的一个圆

可以将上述代码封装在不同的函数里，以增加代码的可读性和复用性。例如：

```
import sys
import pygame

WIDTH = 600
HEIGHT = 400
# 数据：圆的参数
circle_pos = (0,0)
circle_radius = 0
circle_color = (0,0,0)
#初始化游戏窗口
def init_window():
    # 1. 初始化
    pygame.init()          #初始化 Pygame

    #设置窗口的模式，(680，480)表示窗口的(宽度，高度)
    #此函数返回一个用于绘制的 surface 对象(相当于一块画布)
    surface = pygame.display.set_mode((WIDTH, HEIGHT))

    pygame.display.set_caption('Game Engine')       #设置窗口标题
    return surface

#初始化场景数据
def  init_scene():
    global circle_pos,circle_radius,circle_color
    circle_pos = (WIDTH//2,HEIGHT//2)
    circle_radius = 20
circle_color = (255,255,255)
#1. 游戏初始化
def init():
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    surface = init_window()
    init_scene()
    return surface

#2.1 处理(键盘、鼠标等)事件
def processEvent():
    for event in pygame.event.get():          #返回当前的所有事件
        if event.type == pygame.QUIT:         #接收到窗口关闭事件
            return False                      #退出游戏
    return True

# 2.2 更新游戏的数据
def update():
    pass

#2.3 在 surface 对象上绘制并显示场景
def draw(surface):
     pygame.draw.circle(surface, circle_color,circle_pos,circle_radius)
     pygame.display.flip()  #update()

# ------游戏主函数-----
def game_engine():
    surface = init()                  #1. 初始化 Pygame 和游戏数据

     #2. 循环，直到游戏结束
    running = True
    while running == True:
        running = processEvent()      #2.1 处理事件
        update()                          #2.2 更新数据
        draw(surface)                 #2.3 绘制场景

    pygame.quit()                     #3.退出程序

if __name__ == "__main__":
     game_engine()
```


3.7.3 Pong 游戏

1972 年，美国的雅达利(Atari)公司开发了一款模拟两个人打乒乓球的、称为“Pong”的街机游戏，它被认为是游戏机历史的起点。该游戏是在两条线中间有一个点在动，用摇杆的按钮操纵两边的线作为挡板。Atari 公司的两位老板聘用了一个没有任何游戏开发经验的青年 Alcorn 制作该款游戏，Alcorn 全身心地投入到游戏制作之中，两位老板对于 Alcorn 非常欣赏，虽然他们对这款游戏并不十分有信心，但也制作了一个 Pong 样板机，放在酒吧进行测试，第一台 Pong 游戏机被安装在酒吧后就导致酒吧人满为患。Atari 公司的老板看到了巨大的商机，开始扩大生产线。很快，一台台 Pong 游戏机出现在美国的各个角落，掀起了全美的电子游戏风暴。“Pong”把电子游戏的观念第一次带给了大众。火热的“Pong”引来了其他厂商纷纷效仿，而遥远的日本，也受到了“Pong”的鼓舞，日本的娱乐公司 Taito 开发了一款类似的产品“Elepong”，成为了日本的第一款电子游戏。

如图 3-7 所示，是 Pong 游戏的画面，画面中有一个乒乓球(Ball)和两个挡板(Paddle)，乒乓球初始在区域中心，并沿着随机方向运动，两个挡板分别由两个游戏者操作，试图将球击向反方，如成功击回，就得一分，如未能挡住球，就将失去一分，球将从新的随机位置再次沿着随机方向运动。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

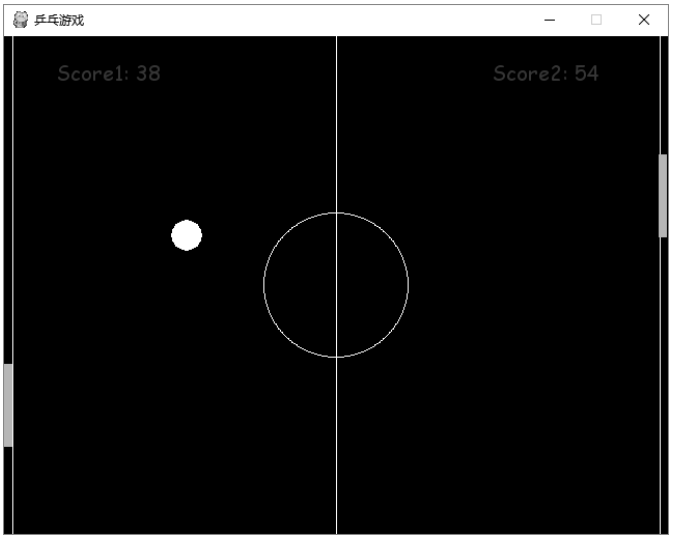


图 3-7　Pong 游戏的画面

1．初始化

游戏中包含一只球(Ball)和左右两个挡板(Paddle)，分别用一个圆和两个矩形表示。球有位置、大小(半径)、颜色、速度等属性，而挡板也有位置、大小(长宽)、颜色、速度等属性。游戏还有记录各自分数的变量。当然，游戏窗口也有长宽、标题、背景颜色等属性。游戏开始时，需要对这些数据初始化：

```
WIDTH = 600                          #窗口宽度
HEIGHT = 400                         #窗口高度

BALL_RADIUS = 15                     #球的半径
ball_pos = [0,0]                     #球的位置
ball_vel = [0,0]                     #球的速度
PAD_WIDTH = 8                        #挡板宽
PAD_HEIGHT = 80                      #挡板高
HALF_PAD_WIDTH = PAD_WIDTH//2
HALF_PAD_HEIGHT = PAD_HEIGHT//2
paddle1_pos = [0,0]
paddle2_pos = [0,0]

paddle1_vel = 0                      #左 Paddle 速度(上下移动的速度)
paddle2_vel = 0
score1 = 0                           #左 Paddle 得分
score2 = 0                           #右 Paddle 得分

#常用颜色 (R,G,B) (红黄蓝)
WHITE = (255,255,255)
RED = (255,0,0)
GREEN = (0,255,0)
BLACK = (0,0,0)
```

在函数 init_scene()中完成这些对象的初始化工作：

```
import random

def init_scene():
    #初始化左右 Paddle(挡板)的属性
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    global paddle1_pos, paddle2_pos, paddle1_vel, paddle2_vel
    global score1, score2
    paddle1_pos = [HALF_PAD_WIDTH, HEIGHT // 2]
    paddle2_pos = [WIDTH - 1 - HALF_PAD_WIDTH, HEIGHT // 2]
    paddle1_vel = [0, 0]
    paddle2_vel = [0, 0]
    score1 = 0
    score2 = 0

    #初始化球的属性
    global ball_pos, ball_vel
    ball_pos = [WIDTH / 2, HEIGHT / 2]
    horizontal = random.randrange(2,4)      #随机生成的水平速度
    vertical = random.randrange(1,3)        #随机生成的垂直速度

    if random.random()>0.5:                 #随机地向左、向右
       horizontal= -horizontal
    if random.random()>0.5:                 #随机地向上、向下
       vertical= -vertical

    ball_vel = [horizontal,-vertical]
```

2．绘制场景

用一个单独的函数 draw(surface)负责绘制场景，其中，surface 代表可以绘制的画布。该函数在 surface 上绘制球和挡板：

```
import pygame, sys
from pygame.locals import *

CICLE_RADIUS  = 70                  #背景中的中心圆半径

def draw(surface):
    global paddle1_pos, paddle2_pos, ball_pos, ball_vel, score1, score2

    #绘制画面背景
    surface.fill(BLACK)             #背景颜色为黑色
    pygame.draw.line(surface, WHITE, [WIDTH // 2, 0],[WIDTH // 2, HEIGHT], 1)
    pygame.draw.line(surface, WHITE, [PAD_WIDTH, 0],[PAD_WIDTH, HEIGHT], 1)
    pygame.draw.line(surface, WHITE, [WIDTH - PAD_WIDTH, 0],[WIDTH - PAD_WIDTH,
         HEIGHT], 1)
    pygame.draw.circle(surface, WHITE, [WIDTH//2, HEIGHT//2], CICLE_RADIUS, 1)


    #绘制挡板 Paddles 和球 Ball
    pygame.draw.circle(surface, WHITE, (int(ball_pos[0]),int(ball_pos[1])),
         BALL_RADIUS, 0)
    pygame.draw.rect(surface, GREEN, (int(paddle1_pos[0]) - HALF_PAD_WIDTH,
         int(paddle1_pos[1]) - HALF_PAD_HEIGHT, PAD_WIDTH,PAD_HEIGHT))
    pygame.draw.rect(surface, GREEN, (int(paddle2_pos[0]) - HALF_PAD_WIDTH,
         int(paddle2_pos[1]) - HALF_PAD_HEIGHT, PAD_WIDTH,PAD_HEIGHT))

    #绘制得分 scores
    drawText(surface,"Score1: "+str(score1),(50,20))
    drawText(surface,"Score2: "+str(score2), (470, 20))

    pygame.display.flip()       #刷新画面

# 辅助函数：绘制文本。参数：文本、位置、字体名和字体大小
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
def drawText(surface,text,pos=(1,1),color=RED,font_name="Comic Sans MS",font_size=20):
    myfont = pygame.font.SysFont(font_name,font_size)
    text_image = myfont.render(text,1,color)
    surface.blit(text_image, pos)
```

执行主函数：

```
if __name__ == "__main__":
    game_engine()
```

就可以看到如图 3-8 所示的静止的游戏场景。

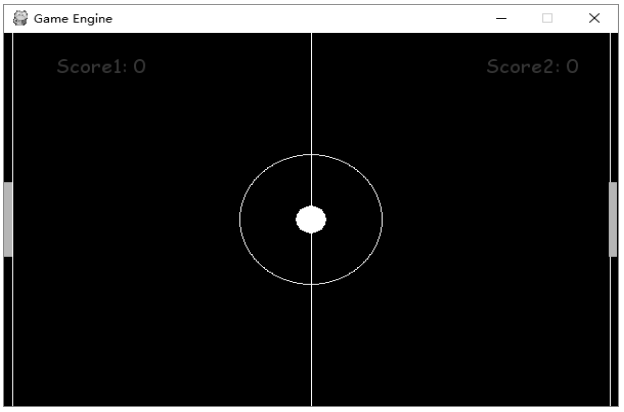


图 3-8　静止的游戏场景

3．让球动起来

在上面的程序中，球和挡板是静止的，这是因为球的位置初始化后就一直没有更新，为了让球动起来，可以根据随机的方向和速度，让球的位置发生变化，可以通过函数 update()来更新球的位置(场景数据)。

```
# 2.2 更新游戏的数据
def update():
    # 更新球
    ball_pos[0] += int(ball_vel[0])
    ball_pos[1] += int(ball_vel[1])

if __name__ == "__main__":
    game_engine()
```

执行以上程序可以看到球一闪而过，超出窗口后就不见了。因此，需要解决如下问题。

- 和上下墙碰撞，球应该反弹回来。
- 和左右挡板碰撞，也应该反弹回来，且击中的一方增加 1 分。
- 越过左右边界，给对方增加 1 分。让球重新出现在中心，并以新的随机速度运动。

代码如下：

```
#初始化球的初始位置和速度
def ball_init():
    global ball_pos, ball_vel # these are vectors stored as lists

    ball_pos = [WIDTH / 2, HEIGHT / 2]
    horizontal = random.randrange(2,4)
    vertical = random.randrange(1,3)

    #表示球向右运动
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            if random.random()>0.5:
                horizontal= -horizontal
            if random.random()>0.5:
                vertical= -vertical

            ball_vel = [horizontal,-vertical]

        # 2.2 更新游戏的数据
        def update():
            global ball_pos, ball_vel # these are vectors stored as lists
            global  score1, score2

            # 更新球
            ball_pos[0] += int(ball_vel[0])
            ball_pos[1] += int(ball_vel[1])
            #与上下墙碰撞，水平速度不变，垂直速度相反
            if ball_pos[1] < BALL_RADIUS or ball_pos[1] > HEIGHT - 1 - BALL_RADIUS:
                ball_vel[0] = ball_vel[0]
                ball_vel[1] = -ball_vel[1]

             # 检测挡板是否和球碰撞
            if ball_pos[0] < BALL_RADIUS + PAD_WIDTH:
                if ball_pos[1] <= paddle1_pos[1] + HALF_PAD_HEIGHT and
                          ball_pos[1] >= paddle1_pos[1] - HALF_PAD_HEIGHT:
                    ball_vel[0] = -(ball_vel[0] * 1.1)   #挡板击中球
                else:                                    #挡板没有挡住球，对方得分
                    ball_init()                          #球重新出现
                    score2 += 1
            elif ball_pos[0] > WIDTH - 1 - BALL_RADIUS - PAD_WIDTH:
                if ball_pos[1] <= paddle2_pos[1] + HALF_PAD_HEIGHT and
                              ball_pos[1] >= paddle2_pos[1] - HALF_PAD_HEIGHT:
                    ball_vel[0] = -(ball_vel[0] * 1.1)
                else:
                    ball_init()                          #球重新出现
                    score1 += 1
```

现在，球能运动起来，且和不同物体碰撞时也会做出适当的反应了。同时，分数也会相应改变。

4．用挡板击打球

然而，挡板还不能运动，这是因为函数 update()没有更新挡板的位置，而且挡板的初始速度也都是 0，挡板的运动需要用户交互控制，如利用鼠标拖动或利用键盘移动。因此，还需要处理用户输入事件，以便决定移动哪个挡板，并在函数 update()中根据挡板速度更新挡板的位置。

5．更新挡板位置

为了更新挡板位置，只要在函数 update()中添加“根据挡板速度，更新挡板的垂直位置”的代码就可以了(碰撞检测已经处理过了)：

```
        #更新挡板的垂直位置
            if paddle1_pos[1] + paddle1_vel[1] > HALF_PAD_HEIGHT and
                      paddle1_pos[1]  + paddle1_vel[1] < HEIGHT - 1 - HALF_PAD_HEIGHT:
                paddle1_pos[1] += paddle1_vel[1]

            if paddle2_pos[1] + paddle2_vel[1] > HALF_PAD_HEIGHT and
                      paddle2_pos[1] + paddle2_vel[1] < HEIGHT - 1 - HALF_PAD_HEIGHT:
                paddle2_pos[1] += paddle2_vel[1]
```

同时，只有移动(如 paddle1_pos[1] + paddle1_vel[1])不超出上下墙时，挡板才运动。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

6.（键盘）事件处理

假设挡板的速度由用户通过上下箭头按键（K_UP 和 K_ DOWN）和字母 w（K_w）和 s（K_s）控制：

```
#键盘按下事件处理函数：更新挡板的垂直速度
def keydown(event):
    global paddle1_vel, paddle2_vel

    if event.key == K_w:
        paddle1_vel = -8
    elif event.key == K_s:
        paddle1_vel = 8
    elif event.key == K_UP:
        paddle2_vel = -8
    elif event.key == K_DOWN:
        paddle2_vel = 8

#键盘弹起事件处理函数：挡板速度重置为 0
def keyup(event):
    global paddle1_vel, paddle2_vel

    if event.key in (K_w, K_s):
        paddle1_vel = 0
    elif event.key in (K_UP, K_DOWN):
        paddle2_vel = 0

#2.1 处理(键盘、鼠标等)事件
def processEvent():
    for event in pygame.event.get():          #返回当前的所有事件
        if event.type == pygame.QUIT:         #接收到窗口关闭事件
            return False                      #退出游戏

        elif event.type == KEYDOWN:
            keydown(event)
        elif event.type == KEYUP:
            keyup(event)

    return True
```

当按下键盘的上下箭头按键或字母 w 与 s 时，可以使左右挡板具有上下移动的速度，当按键弹起时，速度又变为 0。

3.8 实战：线性回归

3.8.1 机器学习

“机器学习”（maching learning）是指用某种学习算法从经验数据中发现规律，再将这个规律用于新的情况的判断、决策或预测。例如，气象预报部门根据以往的经验（大量数据）建立气象预报模型，再利用这个模型对新的天气情况进行预报（预测）；下棋算法根据大量棋盘对局的胜负情况，构建下棋算法模型用于指导下棋；医疗诊断系统可以根据大量病人的各种检查指标和其是否患有癌症的信息，建立一个肿瘤诊断模型，这个模型可以用来对一个新病人做出肿瘤诊断；如果有许多不同年龄人的照片，机器学习也可以从这些照片图像和年龄的对应关系中学习一个模型，来对一张新的照片中的人预测其年龄；同样，如果有一组房屋数据（如房屋面积）和房屋价格的数据，机器学习能够建立一个房屋数据和价格的关系模型，用来预测一个新的房屋的价格。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

人工智能主要分为基于规则的逻辑推理，以及基于统计模型的机器学习。

- 基于规则的逻辑推理的一个典型代表就是专家系统。专家系统主要根据专家的经验提炼出一些语义规则，然后用这些规则进行逻辑推理，因为需要用到专家的经验知识，所以也可以称为第一代机器学习。
- 基于统计模型的机器学习。它主要采用一些统计模型，如支持向量机(Support Vector Machine，SVM)、核方法(Kernel Methods)、随机森林、线性或逻辑回归模型、神经网络模型等表示数据中潜在规律的模型，根据大量的数据样本，学习出某种假设模型(神经网络或SVM 模型)的参数，再用这个模型对新的数据进行预测。基于统计模型的机器学习就是人们常说的机器学习，也可以称为第二代机器学习。

深度学习是基于深度神经网络的机器学习，也是目前取得极大成功的人工智能的核心技术。本节介绍的机器学习中的**线性和逻辑回归**是(深度)神经网络的原子和基础，其求解算法，如梯度下降法，也是(深度)神经网络的核心求解算法。

根据用于学习的数据集中的数据样本是否具有明确的答案，机器学习通常又分为**监督学习**和**非监督学习**。监督学习中，每个样本除数据特征外都有明确的答案。例如，一张人脸照片的数据特征就是图像上所有像素点的颜色信息，而这张照片表示的人的年龄就是答案(或目标)。非监督学习中，数据样本只有数据的特征，没有明确的答案。例如，有一个照片集合，但其中的每张照片没有明确的目标，因此希望找出这个照片集合具有的某些规律。因此，既可以用聚类算法将它们分为男人和女人两组，也可以按照肤色将它们分成不同人种的照片。这种在没有明确答案的数据集中寻找某种规律的学习称为**非监督学习**。

线性回归(如从照片预测年龄或从面积预测房屋价格)和逻辑回归(从医学指标判断是否是肿瘤)都属于监督学习。**线性回归**用于预测一个连续值，即预测的结果是一个连续值(如房屋价格)，**逻辑回归**用于分类，即确定一个数据是几种类别中的哪一类(如是如肿瘤还是非肿瘤)。

3.8.2 假设函数、回归和分类

假如有一组房屋面积及其价格的数据，如表 3-1 所示。

可以绘制出这组数据(面积和价格)，如图 3-9 所示。

表 3-1 房屋面积及其价格

房屋面积 (单位：平方米)	价格 (单位：1000 美元/米^2)
2014	400
1600	330
2400	369
1416	232
3000	540
…	…

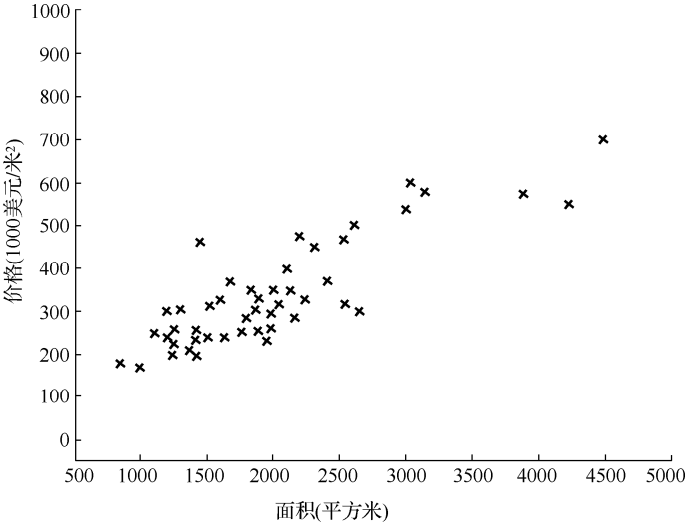


图 3-9 房屋面积及其价格

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

根据这组数据，对于一个面积已知的新房屋，能否预测其价格？这是一个监督学习问题。首先用这组数据学习某个模型，再用这个模型预测新房屋的价格。从数学角度来描述，监督学习实际上就是学习一个数学函数 $y = h(x)$。其中，数据面积称为自变量或特征，用 x 表示，而价格称为因变量或目标变量，用 y 表示，$h(x)$ 是表示 x 和 y 关系的某种数学函数（如线性函数（直线）、二次函数或更加复杂的函数）。

数据集中的每个数据 (x^i, y^i) 称为一个样本。如果要预测的目标变量 y 是一个连续的值，则这种监督学习称为**回归**，如果要预测的目标变量 y 是一个离散的值，则这种监督学习称为**分类**。

假设函数 $h(x)$ 的集合通常是一个无穷集合，但可以用一组未知参数刻画这些函数，如 $y=h(x)=ax+b$，因此不同的 a、b 参数就表示一个不同的函数，回归的目标就是根据一组数据在某种最佳的意义上求出一个参数（如 a、b）确定的假设函数，即确定那些未知参数。

3.8.3 线性回归

1．线性回归的含义

如果表示目标变量 y 和特征 x 之间的假设函数 $h(x)$ 是一个线性函数，那么这种监督学习称为**线性回归**，即线性函数 $h(x)$ 表示的是一个直线。对于一个样本，将其特征 x 代入这个假设函数 $h(x)$ 就得到样本 x 的目标值（预测值）：

$$h_\theta(x) = \theta_0 + \theta_1 x$$

x 和 y 之间的这个线性假设函数对应到二维平面上的图像就是一个直线，不同的参数 θ_0 和 θ_1 对应不同的直线，线性回归就是要求解一个最佳的假设函数（直线），使得所有训练数据集中的样本 $\{x^i, y^i\}$ 和这个最佳的直线最接近。当然，样本点不会正好都位于这个最佳直线上。

查找最佳直线的一种简单的办法是对每个样本 $\{x^i, y^i\}$，用假设函数预测得到的 $h_\theta(x^i)$ 和目标值 y^i 的误差 $(y^i - h_\theta(x^i))^2$ 作为这个样本的误差。线性回归学习的目标是使所有样本的误差之和 $\Sigma(y^i - h_\theta(x^i))^2$ 为最小，这就是人们熟悉的“**最小二乘问题**”，即求使最小二乘代价最小的参数 $\theta = (\theta_0, \theta_1)$。对这个代价函数乘以任意常数都不会改变这个最小二乘问题，一般采用的是如下的代价函数：

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (y^i - h_\theta(x^i))^2$$

其中，m 是样本的数目。线性回归就是求解最佳的 $\theta = (\theta_0, \theta_1)$，以使上述代价函数的值最小：

$$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$$

这个代价函数 $J(\theta_0, \theta_1)$ 是多个未知参数 (θ_0, θ_1) 的函数，即一个多变量的函数，这个多变量的函数的最小值问题（求解 θ）通常有以下两种解法。

(1) 正规方程（normal equation）。

(2) 梯度下降法（gradient descent）。

2．假设函数 $h_\theta(x)$ 的向量表示

上述例子中，样本的特征 x 只有一个特征值，而在实际应用中，一个样本的特征通常包含多个不同的特征值。例如，房屋预测问题，一个房屋的特征值可能包含房屋面积、房间个数等多个特征信息。又如，在疾病诊断中，通常需要检查多个指标。因此，如果一个样本的特征是多个特征值，

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

则可将这些特征值表示为一个向量形式，如一个样本的两个特征值 x_1、x_2，可表示为 $x=(x_1、x_2)$，线性假设函数 $h_\theta(x)$ 可写成：

$$h_\theta(x)=\theta_1+\theta_1x_1+\theta_2x_2$$

为了将 $h(x)$ 写成更简单的统一的向量形式，通常可以人为地给样本添加一个特征值 1，即将 $x=(x_1,x_2)$ 写成 $x=(1,x_1,x_2)$，另外，按照数学向量的习惯写法，一般在进行数学推导时都写成列向量而不是行向量形式，即 $x=(1,x_1,x_2)^{\mathrm{T}}$，$\theta=(\theta_0,\theta_1,\theta_2)^{\mathrm{T}}$。它们的转置就是行向量 $x^{\mathrm{T}}=(1,x_1,x_2)$、$\theta^{\mathrm{T}}=(\theta_0,\theta_1,\theta_2)$。同时，假设函数 $h_\theta(x)$ 可写成向量形式：

$$h_\theta(x)=\theta_0+\theta_1x_1+\theta_2x_2=x^{\mathrm{T}}\theta=\theta^{\mathrm{T}}x$$

3.8.4 多变量函数的最小值、正规方程

假设有一个函数 $J(\theta_0,\theta_1,\ldots,\theta_n)$，目标是 $\min\limits_{\theta_0,\theta_1,\ldots,\theta_n}J(\theta_0,\theta_1,\ldots,\theta_n)$。

根据数学分析（高等数学）知识，函数的最小值点 $(\theta_0,\theta_1,\ldots,\theta_n)$ 的必要条件是函数在该点的导数为 0（对于多变量，即其梯度为 0）。

线性回归的代价函数 $J(\theta)$ 是许多样本的误差累加和，对于其中的一个样本，其关于参数 θ_j 的导数是：

$$\frac{\partial(h_\theta(x^i)-y^i)^2}{\partial\theta_j}=2(h_\theta(x^i)-y^i)\frac{\partial((\theta_0x_0^i+\theta_jx_j^i+\ldots+\theta_nx_n^i)-y^i)}{\partial\theta_j}=2(h_\theta(x^i)-y^i)x_j^i$$

根据导数的加法性质，整个代价函数关于参数 θ_j 的导数就是每个样本关于参数 θ_j 的导数之和的平均：

$$\frac{\partial J(\theta_0,\theta_1,\cdots,\theta_n)}{\partial\theta_j}=\frac{1}{m}\sum_{i=1}^{m}h_\theta(x^i-y^i)x_j^i \tag{1}$$

对于每个 θ_j，令导数等于 0，就得到一个关于 θ_0、θ_1、…、θ_j、…、θ_n 的 n 个方程组成的方程组。

$$\begin{cases}\sum\limits_{i=1}^{m}h_\theta(x^i-y^i)x_0^i=0\\ \vdots\\ \sum\limits_{i=1}^{m}h_\theta(x^i-y^i)x_j^i=0\\ \vdots\\ \sum\limits_{i=1}^{m}h_\theta(x^i-y^i)x_n^i=0\end{cases} \tag{2}$$

可以用矩阵和向量简洁表示上述方程组。令：

$$X=\begin{bmatrix}x_0^1 & x_1^1 & \cdots & x_j^1 & \cdots & x_n^1\\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots\\ x_0^i & x_0^i & \cdots & x_j^i & \cdots & x_n^i\\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots\\ x_0^m & x_1^m & \cdots & x_j^m & \cdots & x_n^m\end{bmatrix}=\begin{bmatrix}x^{1\mathrm{T}}\\ \vdots\\ x^{i\mathrm{T}}\\ \vdots\\ x^{m\mathrm{T}}\end{bmatrix}\quad Y=\begin{bmatrix}y^1\\ \vdots\\ y^i\\ \vdots\\ y^m\end{bmatrix}\quad \theta=\begin{bmatrix}\theta^0\\ \vdots\\ \theta^j\\ \vdots\\ \theta^n\end{bmatrix}$$

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

上述方程组，可以写成矩阵和向量的形式：

$$X^{\mathrm{T}}(X\theta - Y) = 0$$

即 $X^{\mathrm{T}}X\theta = X^{\mathrm{T}}Y$

这个就是所谓的**正规方程**(normal equation)。$X^{\mathrm{T}}X$ 是一个 $n \times n$ 的矩阵，矩阵两边乘其逆矩阵，从而得到方程组的解：$\theta = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}Y$ 。

虽然可以用正规方程方法直接求出线性回归的解 θ，但是如果样本个数 m 很大，或者 θ 向量的参数多，这种矩阵乘积或求矩阵的逆矩阵的计算量很大，效率比较低。因此，一般采用迭代法求方程组的解，其中最常用的方法就是**梯度下降法**(**gradient descent**)：从一个 θ 的初始值出发，沿着梯度方向迭代更新未知参数 θ 。

3.8.5 梯度下降法

梯度下降法(gradient descent algorithm)就是从一个初始的 θ 值，迭代地沿着 J 关于 θ 的梯度的反方向前进(更新 θ 值)，从而不断逼近最佳的 θ 。

- 随机选择一组值作为 θ 的初始值。
- 循环迭代直至结果收敛。

$$\theta_j = \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1, \ldots, \theta_n)}{\partial \theta_j} \tag{3}$$

其中，α 是学习率，表示更新 θ 的速度，太小则收敛缓慢，太大则可能会跳过最佳 θ，导致 θ 值来回振荡。一般来说，这个学习率不是固定的，首先可以取较大值，以加快更新速度，然后逐渐减小，以提高收敛性。

如图 3-10 所示，随着 θ 的更新，其对应的函数值 $J(\theta)$ 也在不断减小(下降)。

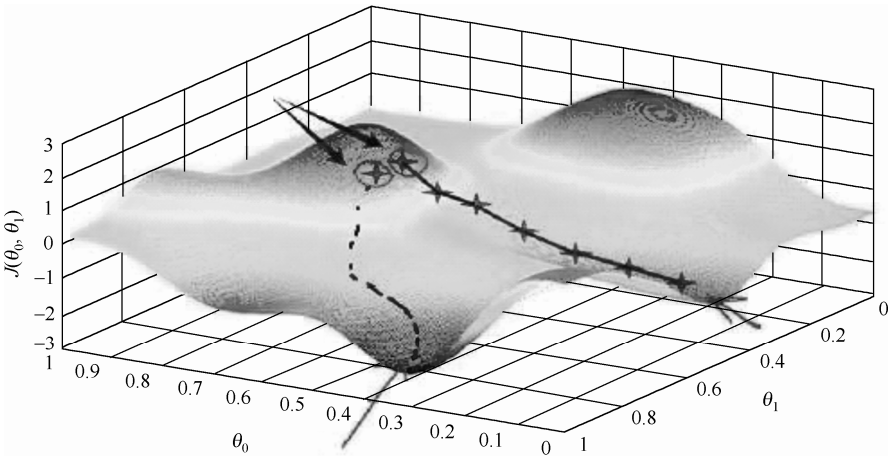


图 3-10　沿梯度反方向更新 θ

因为

$$\frac{\partial J(\theta_0, \theta_1, \ldots, \theta_n)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m} h_\theta(x^i - y^i)x_j^i$$

因此，迭代公式为：

$$\theta_j = \theta_j - \alpha \frac{1}{m}\sum_{i=1}^{m} h_\theta(x^i - y^i)x_j^i \tag{4}$$

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

如图 3-11 所示是 θ 的参数 θ 平面上的一个典型的更新过程，其中，每个圆圈上的 θ 对应的 $J(\theta)$ 都是等值的。

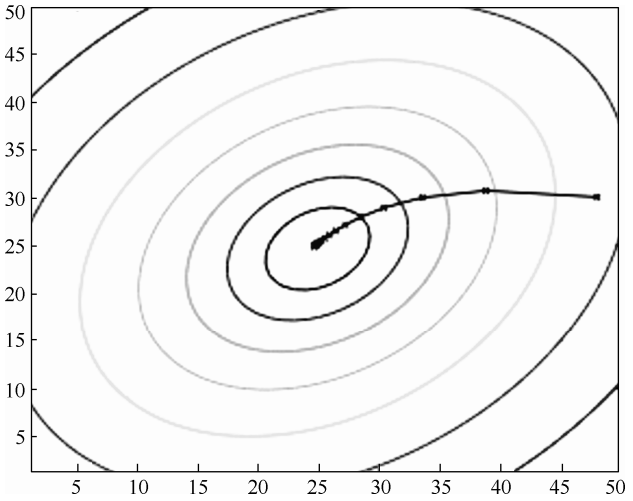


图 3-11　更新过程

3.8.6　梯度下降法求解线性回归问题：模拟数据

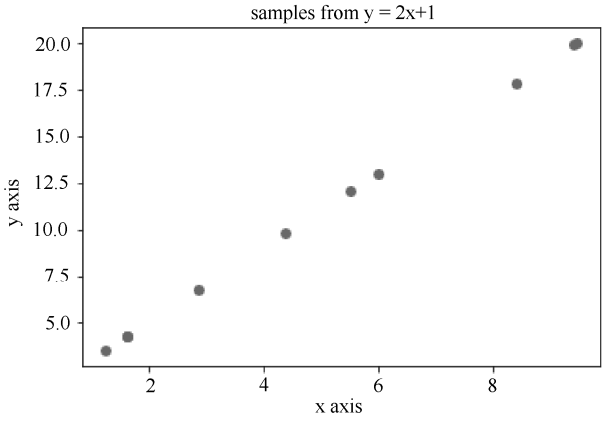
有很多线性回归的文章和代码都是基于第三方库，如多维数组库 numpy 或机器学习库 sklearn 等实现的，特别是机器学习库掩盖了算法的实现细节。本书的线性回归不借助任何第三方库，只利用 Python 内在的数据类型 list，有助于读者更好地理解算法原理和实现细节。

首先，模拟生成一组样本数据，该样本数据是对线性函数 $y=\theta_0+\theta_1 x$ 的随机噪声取样：

```
import random
import matplotlib.pyplot as plt
import time
%matplotlib inline

#m 是样本个数，[0,x_range]是 x 坐标的范围
#对直线 y = 2x+b 进行随机噪声采样
# 返回所有样本的特征的向量 x 和对应目标值的向量 y
def gen_data(k=2, b=1,m=10,x_range=10):
    x = []
    y_true = []
    y = []

    for i in range(m):
        xi = random.uniform(0, x_range)

        x.append([xi])
        yi = k * xi + b
        y_true.append(yi)
        yi_noise = yi + (random.random()-0.5)*2
        y.append(yi_noise)
    return x, y_true

#辅助函数：用于得到矩阵的一列
def column(matrix, i):
    return [row[i] for row in matrix]
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
train_x, train_y = gen_data()
plt.scatter(column(train_x,0), train_y)
plt.xlabel('x axis ')
plt.ylabel('y axis ')
plt.title(u'samples from y=2x+1')
```



其中，使用 matplotlib 库的函数 plt.scatter()将样本特征的向量 x 和对应目标值的向量 y 传递给线性函数，就可以显示这些样本点(样本的特征值 x_i 和目标值 y_i 作为平面上的点)的图像，二维平面上的采样点如图 3-12 所示。

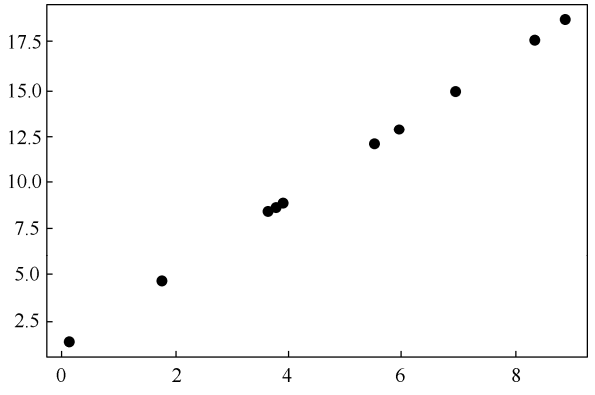


图 3-12　二维平面上的采样点

在下列的线性回归的实现程序中，编写了如下一些函数。

- plot_line()：绘制一个 θ 值对应的假设函数的直线图像。
- h_theta_x()：对一组样本特征 x，计算它们的假设函数 $h_\theta(x)$ 的值。
- cost_function()：计算代价函数 $J(\theta)$ 及其梯度的值。
- gradient_descent()：梯度下降算法更新 θ。
- linear_regression()：传入初始样本数据，调用函数 gradient_descent()求解 θ。

其中，函数 gradient_descent()是梯度下降法的实现，在 for 循环中，首先调用函数 cost_function()，根据样本数据 $\boldsymbol{X}$、$\boldsymbol{Y}$ 和当前的 θ 值，计算代价函数值及代价函数关于 θ 的梯度，然后用公式(3)更新这个 θ 值，经过一定次数的迭代后，再次输出代价函数值，以检查代价函数是否一致在下降，经过 iterations 次迭代后停止迭代，并认为已经收敛(当然还有其他更好的检查迭代是否收敛的方法)。

代码如下：

```
def plot_line(theta,x_range = 10):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    X = []
    predictions = []
    for x in range(x_range):
        X.append(x)
        predictions.append(theta[0]+theta[1]*x)
    plt.plot(X[:], predictions, linewidth=2)

def h_theta_x(X, theta):
    m = len(X)
   # n = len(theta)
    n = len(X[0])
    h = [0]*m
    for i in range(m):
        for j in range(n):
            h[i] +=  X[i][j]*theta[j]
    return h

def cost_function(X, Y, theta):
    m = len(Y)
    n = len(theta)

    f = 0
    g = [0]*n
    h = h_theta_x(X,theta)
    h_y =   [x-y for x,y in zip (h,Y) ]
    f = sum([ x*x for x in h_y])/(2*m)

    for j in range(n):
        for i in range(m):
            g[j] += h_y[i]*X[i][j]
        g[j] /= m

    return f,g

DEBUG_PLOT = True
DEBUG_PLOT_NUM  = 100

def gradient_descent(X, Y, theta, alpha, iterations):
    cost_history = [0] * iterations
    m = len(Y)

    for i in range(iterations):
        cost,gradient = cost_function(X, Y, theta)
        cost_history[i] = cost
        theta = [theta_-alpha*g for theta_,g in zip (theta,gradient) ]
                                    #theta = theta - alpha * gradient
        if i % DEBUG_PLOT_NUM == DEBUG_PLOT_NUM-1:
            print(i,cost)                       #输出迭代次数和代价
            if DEBUG_PLOT:                      #绘制当前 theta 对应的直线
                plot_line(theta)
                time.sleep(1)

    return theta, cost_history

def linear_regression(X,Y,alpha = 0.003, num_iter = 1000):
    n = len(X[0])
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    theta = [0]*n
    theta_opt, cost_history = gradient_descent(X, Y, theta, alpha, num_iter)
    return theta_opt,cost_history

if __name__ == '__main__':
    plt.scatter(column(train_x,0), train_y)
    X = [ [1]+x for x in  train_x]
    theta,cost_history = linear_regression(X,train_y)
    print(theta)
    plt.show()
    plt.plot(cost_history)
```

执行上述程序，输出经过一定次数迭代后的代价函数值，最后输出最终的θ值：

```
99 38.27776007340227
199 20.292214206468028
299 10.774352448594009
399 5.737437976769053
499 3.0717615328519536
599 1.6609030044211557
699 0.9140729866300061
799 0.5186363683392713
899 0.3091522757484421
999 0.19807237169564385
[0.32045613586307387, 2.011517325537776]
```

如图 3-13 所示，通过绘制θ值对应的假设函数直线，有助于人们直观地观察迭代是否不断收敛目标θ值。

如图 3-14 所示，可以通过绘制迭代过程中的代价函数值，直观地观察迭代是否收敛，这是最常用的检查是否收敛的方法。可以看到，当学习率设置为 0.0001 时，经过约 600 次迭代就基本收敛了。因此，可以调整这个学习率，选择一个最佳的学习率，保证足够收敛速度和解的准确性。如果发现梯度下降法不收敛，则除调整学习率外，更应该检查梯度的计算是否正确。为此，可以根据导数的定义，即导数是函数的变化率，用如下公式估计梯度：

$$\frac{\partial J}{\partial \theta}=\lim_{\epsilon\to 0}\frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$$

将这个估计的数值梯度和分析梯度进行比较，当ϵ很小时，二者应该很接近。

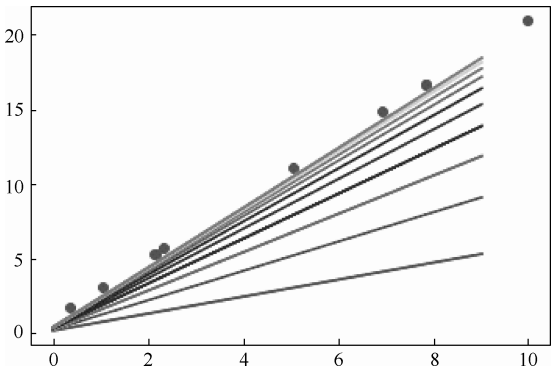


图 3-13　θ值逐渐收敛的直线

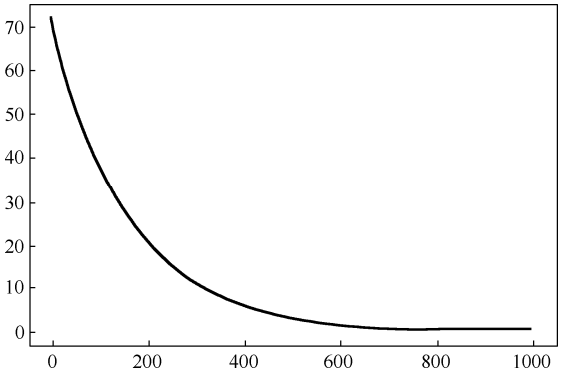


图 3-14　代价函数值不断下降(算法逐渐收敛)

为此，可以编写如下代码来检查分析梯度的计算是否正确。

```
import math
def J(X, Y,theta):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        m = len(X)
        h = h_theta_x(X,theta)
        h_y =   [x-y for x,y in zip (h,Y) ]
        f = sum([ x*x for x in h_y])/(2*m)
        return f
    #估算数值梯度
    def computeNumericalGradient(X,Y, theta, epsilon=1e-7):
        n = len(theta)
        grad_approx = [0]*n
        for j in range(n):
            thetaplus = theta[:]
            thetaplus[j] += epsilon                 # Step 1
            thetaminus = theta[:]
            thetaminus[j] -= epsilon                # Step 2
            J_plus = J(X, Y,thetaplus)              # Step 3
            J_minus = J(X, Y,thetaminus)            # Step 4
            grad_approx[j] = (J_plus - J_minus) / (2 * epsilon)       # Step 5
        f,grad = cost_function(X,Y,theta)

        print(grad)
        print(grad_approx)
        diff  =  [g-g_approx for g,g_approx in zip(grad,grad_approx) ]
        difference = sum([ math.fabs(x) for x in diff ])/n
        if difference < 1e-7:
            print("The gradient is correct!",difference)
        else:
            print("The gradient is wrong!",difference)
    #用下面的代码检查数值梯度计算是否准确
    X = [ [1]+x for x in  train_x]
    theta = [0.5,1.5]   # theta 的值应接近真正的 theta 值
    computeNumericalGradient(X,train_y,theta)
```

输出：

```
[-2.952050983202688, -17.909524322700243]
[-2.9520509725600164, -17.90952432578763]
The gradient is correct! 6.865028989722077e-09
```

3.8.7 批梯度下降法

当数据集很大时，每次梯度更新需要用所有数据计算代价和梯度，一方面比较耗时，另一方面如果数据集很大，则内存可能无法全部存放，因此，通常采用批梯度下降法(batch gradient descent)进行梯度更新，也就是每次只用部分数据来计算代价和梯度并更新参数。假设数据集样本总数是 M，每次从 M 个样本中随机选取 m(m<<M)个样本做梯度更新。

假如上述的函数 gradient_descent()是针对 m 个样本的梯度更新，则可以在其外层再定义一个函数 batch_gradient_descent ()用来随机选取 m 个样本，然后交给函数 gradient_descent()用这 m 个样本进行梯度更新。例如：

```
    def batch_gradient_descent(X, Y, theta, alpha, iterations=300,m = 5,batch_
                iterations = 2):
        total_cost_history = []
        M = len(Y)

        for i in range(batch_iterations):
            select = random.sample(range(M), m)
            X_m = [X[i] for i in select]
            Y_m = [Y[i] for i in select]
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            theta, cost_history = gradient_descent(X_m,Y_m,theta, alpha, iterations)
            total_cost_history +=  cost_history
        return theta,total_cost_history
    def linear_regression_batch(X,Y,alpha = 0.003, iterations = 100,sample_m  =5,
                batch_iterations = 11):
        n = len(X[0])
        theta = [0]*n
        theta_opt, cost_history = batch_gradient_descent(X, train_y, theta, alpha,
                 iterations,sample_m,batch_iterations)
        return theta_opt,cost_history

    if __name__ == '__main__':
        plt.scatter(column(train_x,0), train_y)
        X = [ [1]+x for x in  train_x]
        theta,cost_history = linear_regression_batch(X,train_y,0.003,200,7,6)
        print(theta)
        plt.show()
        plt.plot(cost_history)
```

输出：

```
99 10.751378063370755
199 1.6432006948356548
299 0.3163534108709256
399 0.12156124827204276
499 0.09149871631049332
99 0.10200797917285363
199 0.09682687364162433
299 0.09353656836730527
399 0.09084090646257989
499 0.08836367021777085
99 0.07520929967296533
199 0.07336062312433748
299 0.07163814618244362
399 0.06997217015541814
499 0.06834812644469475
99 0.05630680825289037
199 0.05510176722413733
299 0.05401488881705586
399 0.052960651464657775
499 0.05192834646707161
99 0.0421704358704574
199 0.04129365041994414
299 0.04059908219387134
399 0.03994314424554974
499 0.03930221423838593
99 0.055819024765825176
199 0.053327471839137545
299 0.05163827648794717
399 0.050285114549252115
499 0.0490813575135783
[0.5002588692540729, 2.101742321889468]
```

如图 3-15 所示，随着 θ 值不断收敛，对应直线逼近目标直线。如图 3-16 所示，代价函数值不断下降，在 200 次左右基本收敛。

如果数据量很大，则采用批梯度下降法可以大大提高收敛速度和计算量。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

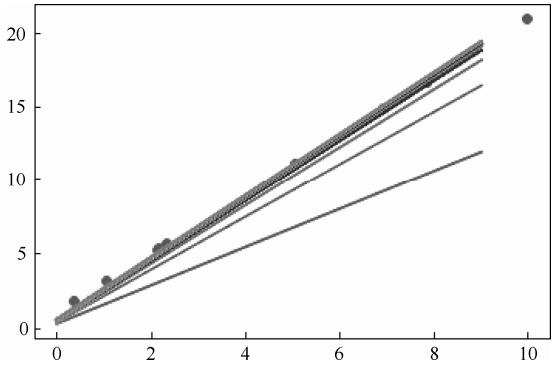


图 3-15　批梯度下降法：θ 值逐渐收敛的直线

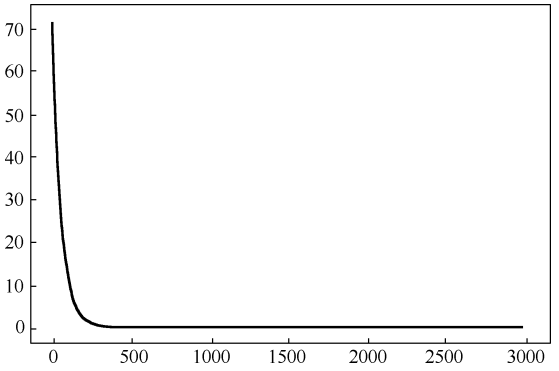


图 3-16　批梯度下降法：代价函数值不断下降(算法逐渐收敛)

3.8.8　房屋价格预测

以下使用的是斯坦福大学吴恩达的机器学习课程中房屋预测问题的数据，这个数据中有 47 个样本，每个样本包含面积、房间数、价格，即样本的特征是面积、房间数，而目标是价格。数据通常存储在文件中，因为未介绍文件读写，这里直接将数据放在一个 list 变量中：


```
house_data = [
2104,3,399900,
1600,3,329900,
2400,3,369000,
1416,2,232000,
3000,4,539900,
1985,4,299900,
...]
```

为使用前面的线性回归代码，将这个数据分成特征和目标两个 list 变量：

```
train_x = []
train_y = []
for i in range(47):
    j = 3*i
    train_x.append([house_data[j],house_data[j+1]])
    train_y.append(house_data[j+2])
#测试是否正确
for i in range(5):
    print(train_x[i],train_y[i])
```

输出：

```
[2104, 3] 399900
[1600, 3] 329900
[2400, 3] 369000
[1416, 2] 232000
[3000, 4] 539900
```

3.8.9　样本特征的规范化

由于一个样本的不同特征(面积、房间数)在数值上具有不同的尺度，如面积的值很大，而房间数很小，因此如果直接用这些特征值进行机器学习，则学习算法会严重倾向于尺度大的特征(如本例中的面积)。为了使不同特征具有同等的作用，需要对这些特征进行规范化，即将它们变换到同样的数值范围里(如[0,1]或[−1,1]范围里)。对样本特征规范化(normalization by standard deviation)的过程很简单，首先需要计算所有样本关于这个特征的平均值，再计算所有样本的这个特征围绕平均值的偏移程度(标准差)，最后将所有样本的这个特征减其平均值再除标准差：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

$$x \leftarrow= \frac{x - \text{mean}(x)}{\text{stddev}(x)}$$

例如，有一组特征{−5, 6, 9, 2, 4}，其平均值 Mean 为：

```
Mean = (-5+6+9+2+4) / 5 = 3.2
```

将所有特征的值减这个平均值得到偏差，并计算这些偏差的平方：

```
(-5-3.2)2 = 67.24
(6-3.2) 2 = 7.84
(9-3.2) 2 = 33.64
(2-3.2) 2 = 1.44
(4-3.2) 2 = 0.64
```

然后，可以计算出标准差 Stddev：

```
Stddev= sqrt ((67.24 + 7.84 + 33.64 + 1.44 + 0.64) / 5) = 4.71
```

最后，所有特征的值的偏差除这个标准差，得到规范化后的特征：

```
x => (x - Mean) / Stddev
-5 => (-5 - 3.2) / 4.71 = -1.74
6 => (6 - 3.2) / 4.71 = 0.59
9 => (9 - 3.2) / 4.71 = 1.23
2 => (2 - 3.2) / 4.71 = -0.25
4 => (4 - 3.2) / 4.71 = 0.17
```


下面是三个辅助函数，用于对样本的某个特征(某一列)进行规范化：

```
import math

def mean(X, column):
    n = len(X)
    return sum(x[column] for x in X)/n

def standard_deviation(X, column):
    n = len(X)
    mean_ = mean(X, column)
    return math.sqrt(sum((x[column]-mean_)**2 for x in X)/n )

def Normalization(X, column):
    mean_ = mean(X,column)
    deviation = standard_deviation(X,column)
    for x in X:
        x[column]  = (x[column]-mean_)/deviation
```

现在，对 $\boldsymbol{X}$ 的两个特征用上述的函数进行规范化：

```
# 显示几个样本数据，查看规范化是否正确
for i in range(5):
    print(train_x[i])
Normalization(train_x,0)
Normalization(train_x,1)
for i in range(5):
    print(train_x[i])
```

输出：

```
[2104, 3]
[1600, 3]
[2400, 3]
[1416, 2]
[3000, 4]
[0.13141542202104753, -0.22609336757768828]
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
[-0.5096406975906851, -0.22609336757768828]
[0.5079086986184144, -0.22609336757768828]
[-0.743677058718778, -1.5543919020966084]
[1.2710707457752388, 1.1022051669412318]
```

现在，可以用前面的线性回归的方法对房屋特征及其价格数据进行线性回归，得到一个最佳的假设线性函数：

```
#theta, alpha, iterations=500,m = 5,batch_iterations = 10
DEBUG_PLOT = False

theta = [0]*3
alpha = 0.01   # learning rate
iterations = 400
sample_m = 12
batch_iterations = 5

X = [ [1]+x for x in  train_x]

#theta_opt, cost_history = batch_gradient_descent(X, train_y, theta, alpha,
        iterations,sample_m,batch_iterations)
theta_opt, cost_history = gradient_descent(X, train_y, theta, alpha, iterations)
print(theta_opt)
plt.plot(cost_history)
```


输出：

```
299 2291774852.9913735
[334302.06399327697, 99411.4494735893, 3267.012854065695]
```

如图 3-17 所示，对这个房屋价格数据，代价函数值不断下降，算法逐渐收敛。

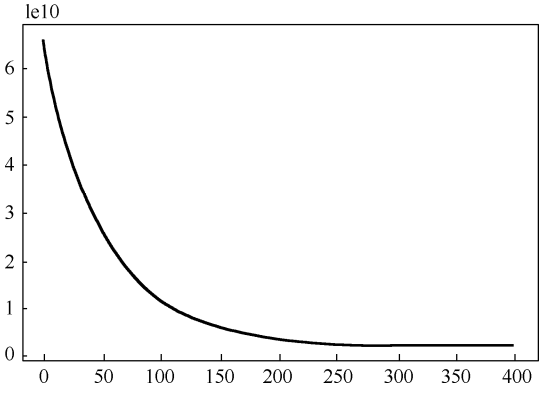


图 3-17　代价函数值不断下降(算法逐渐收敛)

3.8.10　利用预测模型预测房屋价格

得到了房屋特征值和价格的预测模型，就可以用这个模型预测一个新的房屋的价格。例如，预测一个新房屋(1650 平方米，房间数是 3)的价格：

```
theta = theta_opt
# 面积和房间数
area = 1650
bed_rooms = 3
# 预测价格(y 值)
x = (1,(area-mean_1)/deviation_1,(bed_rooms-mean_2)/deviation_2)     #规范化
predictedSellingPrice = x[0]*theta[0]+ x[1]*theta[1]+ x[2]*theta[2]
                                                    #根据假设函数计算预测价格
print('${:,.2f}'.format(predictedSellingPrice))     #打印预测的价格
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
$289,221.55
```

3.9 习题

1. 下列关于函数的定义正确的是(　　)。

   A. 函数是用于创建对象的　　B. 函数使代码执行更快

   C. 函数是完成特定工作的代码片段　　D. 上述陈述都对

2. 下列代码的输出结果是什么？(　　)

```
def printJob(name,job):
    print(name)
    print("你的工作是：",job)
printJob("hwdong","教小白精通编程")
```

3. 如果函数中没有 return，则函数返回的是(　　)。

   A. 0　　B. None

   C. 任意整数　　D. Python 的函数必须包含 return 语句

4. 下列是根据 BMI 指数判断一个人是否超重(BMI 指数小于 25，表示未超重，否则就超重)的函数。请完善此函数，并编写代码，要求从键盘输入身高和体重，并测试这个函数。

```
def overweight_by_BMI(name ,height,weight):
     bmi = ?     #用 bmi 变量表示 BMI 指数
     if ? :
        print(name,"没有超重")
     else
        print(name,"超重了")
     print("你的 BMI 指数是:",bmi)
     return bmi
```

5. 请判断下列代码的输出结果是什么？并请判断下列代码是否有错误，如有错误，则请指出错误并改正。(　　)

```
X="global"
def fun(x)
    x=x*2
    print(x)
fun(X)
print(X)
```

6. 编写一个非递归的函数用于接收一个整数参数 *n*，并返回这个 *n* 对应的斐波那契(Fibonacci)序列，可以用一个 list 保存这个序列，如 Fib(100)返回的 Fibonacci 序列是：

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

然后，从键盘输入一个整数，调用这个函数并打印结果。

7. 定义一个求一元二次方程的函数，方程的系数作为该函数的三个参数，要求返回一个无解、一个解或两个解的结果，并从键盘输入方程系数，以测试函数是否能够正确求解。

8. 编写一个函数，计算传入的字符串中的数字、字母、空格和其他字符的个数。

9. 编写一个函数，将传入的 list 对象的元素每间隔三个元素选择一个元素，构造并返回一个 tuple 对象。例如：

```
输入：[1,2,3,4,5,6,7,8,9]
输出：(1,4,7)
```

10. 将一个英文语句以单词为单位逆序排放。例如，“I am a boy”，逆序排放后为“boy a am I”。假设所有单词之间用一个空格隔开，语句中除英文字母外，不再包含其他字符。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

11．重新实现第 2 章习题中的学生成绩分析程序，用一系列函数完成单独的功能。例如，编写专门的函数用于输入一个学生的信息。

12．编写一个函数，该函数不调用内置的排序函数就可对三个数进行排序和输出，要求该函数附带一个默认参数 reverse=False，表示默认从小到大排序，当该形参对应的实参为 True 时，则从大到小排序。

13．编写一个可以求多个数的乘积函数 Product()，可以接收一个或多个数作为参数。运行示例如下：

```
print('Product(3)', Product(3))
print('Product(3,4)', Product(3,4))
print('Product(3,4,5)', Product(3,4,5))
…
```

14．编写一个函数 Student()，除学生名外，还接收可变数目信息和关键字信息，并将学生信息以 list 对象形式返回。运行示例如下：

```
Student('Li',23)
Student('Li',21,'男',class='计科 3 班','city'= 'Shanghai','phone'='12381233451')
```

15．递归函数是(　　)。

A．在程序中调用所有函数的函数　　B．调用自身的函数

C．调用除自身以外的所有函数的函数　　D．Python 没有递归函数的概念

16．编写一个函数，如果输入一个整数 *n*，则输出 *n* 行的 Pascal 三角形，如下所示。

```
              1
            1   1
          1   2   1
        1   3   3   1
      1   4   6   4   1
    1   5   10  10  5   1
```

17．下列代码的输出结果是什么？(　　)

```
result = lambda x: x * x
print(result(5))
```

A．lambda x: x*x　　B．10　　C．25　　D．5*5

18．下列代码的输出结果是什么？(　　)

```
numbers = [1, 3, 6]
newNumbers = tuple(map(lambda x: x , numbers))
print(newNumbers)
```

A．[1, 3, 6]　　B．(1, 3, 6)　　C．[2, 6, 12]　　D．(2, 6, 12)

19．改写“猜数字游戏”程序，用随机数模块的随机数函数随机生成一个“待猜的数字”。

20．用普通函数和 Lambda 函数定义下列函数。

(1)输入两个数，返回这两个数的平方和的根。

(2)可接收任意几个数，并计算这些数的平均值。如果没有输入任何数，则平均值为 0。

(3)输入一个字符串，返回一个不包含重复字符的字符串。

21．利用函数 map()，把用户输入的不规范的英文名字变为首字母大写、其他小写的规范名字。例如，输入[‘adam’,‘LISA’,‘barT’]；输出[‘Adam’,‘Lisa’,‘Bart’]。

22．假如要输出 math 模块中的常数 pi,下列代码正确的是(　　)。

A．print(math.pi)　　B．print(pi)

C．from math import pi
print(pi)　　D．from math import pi
print(math.pi)

23．下列哪个运算符用于从一个包导入一个模块？(　　)

A．“.”运算符　　B．“*”运算符　　C．“->”运算符　　D．“,”运算符

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 4 章　内置数据类型

4.1　数值

4.1.1　int、float、complex、bool

Python 的数值类型有 int、float、complex、bool。其中，int 表示整数，其长度只要不超出计算机内存即可；float 表示浮点数，小数部分最多 15 位。int 和 float 型数值根据是否有小数点来区分，如 2 是 int 型，而 2.0 是 float 型。而 complex 型数值表示成 x+yj 的形式。bool 表示逻辑值，bool 型数值只有 True（真）和 False（假）两个值，如比较或逻辑运算的结果就是一个 bool 型的数值。例如：

```
print(type(2))
print(type(2.0))
print(type(2+3.9j))
print(type(True))
print(type(3>2))
```

输出：

```
<class 'int'>
<class 'float'>
<class 'complex'>
<class 'bool'>
<class 'bool'>
```

可以用一个内置函数 isinstance()判断一个对象是否为某种类型的对象。例如：

```
print(isinstance(2,int))
print(isinstance(2.0,int))
print(isinstance(2+3.9j,complex))
print(isinstance(False,bool))
```

输出：

```
True
False
True
True
```

日常生活中的数值通常用十进制表示，但计算机中用二进制、十六进制和八进制表示。Python 可通过在数的前面加前缀 0b 或 0B 来表示二进制数，加前缀 0o 或 0O 来表示八进制数，加前缀 0x 或 0X 来表示十六进制数。例如：

```
print(0b0110100101)
print(0O703516)
print(0X7F3BA)
```

输出：

```
421
231246
521146
```

可以用内置函数 bin()、oct()、hex()分别得到一个数的二进制、八进制、十六进制对应的字符串。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

例如：

```
print(bin(412))
print(oct(412))
print(hex(412))
```

输出：

```
0b110011100
0o634
0x19c
```

4.1.2 类型转换

1. 隐式类型转换

不同类型的数值参与运算时，会自动进行隐式类型转换，一般将 int 型数值转换为 float 型数值，或 float 型数值转换为 complex 型数值。例如：

```
a =  2+3.14
print(a)
print(type(a))
```


上述代码的“2+3.14”会自动将 2 首先转换为 float 型数值，再进行加法运算。

输出结果：

```
5.140000000000001
<class 'float'>
```

执行下列代码：

```
a = 2+3.14+(5+6j)
print(a)
print(type(a))
```

输出：

```
(10.14+6j)
<class 'complex'>
```

2. 显式类型转换

可以用内置函数 int()、float()、complex()分别将其他类型的数值强制进行显式类型转换，成为 int、float、complex 型数值。

```
a = int('235')
print(a)
a  = int(3.14)
print(a)

b = float(2)
print(b)

b = complex(b)
print(b)
b = complex('3+5j')
print(b)
```

输出：

```
235
3
2.0
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
(2+0j)
(3+5j)
```

但将一个不是数值类型的字符串强制转化为数值类型则会出现错误，例如：

```
float('h')
```

会抛出异常：

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-b2b9f653472b> in <module>()
----> 1 float('h')

ValueError: could not convert string to float: 'h'
```

当然也可以用函数 str()将类型数值转换为字符串 str 类型数值。例如：

```
print(str(2))
print(str(3+5j))
type(str(2))
```

输出：

```
2
(3+5j)
str
```


4.1.3 浮点数的精度问题

浮点数在计算机中是用二进制近似表示的，因为计算机只能理解二进制(0 和 1)，所以大部分数值的小数部分无法准确地存储在计算机中。例如：

```
print( (2.3-2.2)==0.1)
```

输出：

```
False
```

数学上，十进制的浮点数 2.3–2.2 和 0.1 是完全相等的，但由于计算机只能用二进制近似表示十进制小数，因此，它们是不相等的。例如：

```
print(2.3-2.2)
print(0.1)
```

输出：

```
0.09999999999999964
0.1
```

另外，由于 float 型数值的小数部分最多 15 位，对于一些关键的应用，如金融、科学计算等需要更高精度的场合，Python 提供了更高精度的 Decimal 类。例如：

```
import decimal
print(0.1)
print(decimal.Decimal(0.1))   #更高精度的 0.1
```

输出：

```
0.1
0.1000000000000000055511151231257827021181583404541015625
```

Python 还提供了用于分数计算的 Fraction 类。例如：

```
from fractions import Fraction as F
print(F(1,3))          #表示分数 1/3
print(1 / F(5,6))      #表示分数 6/5
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print(F(1,3) + F(3,4))
```

输出：

```
1/3
6/5
13/12
```

4.1.4 数值计算的函数

可以用 Python 内置的函数做一些数值计算。例如：

```
print(abs(-3.4))          #绝对值
print(min(3.4,2.8))       #最小值
print(max(3.4,2.8))       #最小值
print(pow(0.3,4))         #0.3 的 4 次方
print(round(2.8))         #取最接近的整数
print(round(2.3))
print(round(-2.3))
```

输出：

```
3.4
2.8
3.4
0.0081
3
2
-2
```


4.1.5 数学模块

math 模块提供对 C 语言标准库定义的数学函数的访问。该模块使用浮点值进行复杂的数学运算，包括对数和三角函数的运算。这些功能不能用于复数，如果需要支持复数，则需使用另一个数学模块——cmath 模块中相同名称的函数。

1. 特殊的常量

特殊的常量包括 π、e、nan(非数值)和 inf(无穷大)等。例如：

```
import math
print('  π: {:.30f}'.format(math.pi))
print('  e: {:.30f}'.format(math.e))
```

输出：

```
π: 3.141592653589793115997963468544
e: 2.718281828459045090795598298428
```

2. 异常值

浮点计算可以产生两个异常值：inf 和 nan，也可能抛出一个 OverflowError 异常。

(1) 当浮点数超过 float 类数值表示范围时，产生的结果是 inf(无穷大)。例如：

```
x = 10.0 ** 200
y = x*x
print(y,'\t',math.isinf(x*x))
```

输出：

```
inf   True
```

(2) 并不是所有溢出的值都用 inf 表示。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
x = 10.0 ** 200
x**2
```

在执行 x**2 时会抛出异常：“OverflowError: (34, 'Result too large')”。数值计算的结果是 inf 还是抛出溢出异常 OverflowError 是由底层的 C Python 决定的。

(3) 如果除 inf 的结果是未定的，则结果是 nan（未定的数值）。因为 nan 不能和其他值进行比较，所以检查 nan 只能使用函数 isnan()。例如：

```
import math
x = (10.0 ** 200) * (10.0 ** 200)
y = x / x
print(y,'\t', math.isnan(y))
```

输出：

```
nan   True
```

可以使用函数 isfinite()来检查一个数值是常规数值还是特殊值 inf 或 nan。

```
import math
for f in [math.pi, math.e, math.inf, math.nan]:
   print('({:5.2f} {!s})'.format(f, math.isfinite(f)),end = ',')
```

输出：

```
( 3.14 True),( 2.72 True),(  inf False),(  nan False),
```


3. 浮点数的相等比较

由于计算机不能精确地表示浮点数，直接比较两个浮点数相等是危险的。例如：

```
a= 1.0-0.99
b = 100-99.99
print(a==b)
```

输出：

```
False
```

比较两个浮点数是否相等实际上就是看它们差的绝对值是否足够小，Python 的 math 模块提供了函数 isclose()，其格式是：

```
isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
```

默认形参 rel_tol 和 abs_tol 分别表示相对误差和绝对误差，用于比较两个浮点数是否相等。函数 isclose()等价于下列代码：

```
abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)
```

例如，用于相对误差比较 abs(a-b) <= rel_tol * max(abs(a), abs(b))时：

```
a = 1.00
b = 0.90
print(abs(a-b) <= rel_tol * max(abs(a), abs(b)),end=',')
print(math.isclose(a, b, rel_tol=rel_tol))
a = 0.10
b = 0.09
print(math.isclose(a, b, rel_tol=rel_tol),end=',')
print(abs(a-b) <= rel_tol * max(abs(a), abs(b)))
```

输出：

```
True,True
False,False
```

例如，用于绝对误差比较 abs(a-b) <= abs_tol 时：

```
abs_tol = 1e-07
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
a = 1.00
b  =1.0000001
print(abs(a-b) <= abs_tol,end=',')
print(math.isclose(a, b, abs_tol=abs_tol))
a = 1.00
b  =1.00000001
print(abs(a-b) <= abs_tol,end=',')
print(math.isclose(a, b, abs_tol=abs_tol))
```

输出：

```
False,False
True,True
```

4．浮点数转换为整数

有三个函数可以将浮点数转换为整数，函数 trunc()截断浮点数的小数部分，函数 floor()将浮点数转换为比它小的最大整数，函数 ceil()将浮点数转换为比它大的最小整数。例如：

```
HEADINGS = ('i', 'int', 'trunk', 'floor', 'ceil')
print('{:^5} {:^5} {:^5} {:^5} {:^5}'.format(*HEADINGS))
fmt = '{:5.1f} {:5.1f} {:5.1f} {:5.1f} {:5.1f}'

VALUES = [-1.5, -1.8, 1.2]
for i in VALUES:
   print(fmt.format( i,int(i), math.trunc(i), math.floor(i), math.ceil(i) ))
```


上述代码中使用了 str 类型的 format()方法，{}表示一个占位符，输出时要用 format()方法的实参替换这个占位符；{:^5}表示以宽度为 5 且中间对齐的格式输出 format()方法里的实参；{:5.1f}表示以宽度为 5、小数点后 1 位的浮点数输出格式。

输出：

```
  i    int  trunk floor ceil
 -1.5  -1.0  -1.0  -2.0  -1.0
 -1.8  -1.0  -1.0  -2.0  -1.0
  1.2   1.0   1.0   1.0   2.0
```

5．平方根、指数、对数

函数 sqrt(x)用于专门计算一个数 x 的平方根。例如：

```
print(math.sqrt(3))
```

输出：

```
1.7320508075688772
```

幂函数 pow(x,y)计算 x 的 y 次方，其功能类似 x**y，但 pow(x,y)可以保证按照浮点数运算，而 x ** y 只可以返回一个 int 或一个 float。例如：

```
x = 2.1
y = 3.2
print(x**y)
print(math.pow(x, y))              #计算 x 的 y 次方
print(2**3)
print(math.pow(2,3))
```

输出：

```
10.74241047739471
10.74241047739471
8
8.0
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

对于以自然常数为底的指数函数 exp()，exp(x)计算 e**x 的结果比 math.pow(math.e, x)更准确，对于接近 0 的 x，函数 expm1()用于计算 e**x–1，比直接用函数 exp(x)–1 更准确。例如：

```
x = 2
fmt = '{:.20f}'
print(fmt.format(math.e ** 2))
print(fmt.format(math.pow(math.e, 2)))
print(fmt.format(math.exp(2)))   #exp(x)比 pow(math.e, x)更准确

x = 0.00000000000000000000000001
print(math.exp(x) - 1)
print(math.expm1(x))           #expm1()用于计算 e**x-1，比直接用 exp(x)-1 更准确
```

输出：

```
7.38905609893064951876
7.38905609893064951876
7.38905609893065040694
7.3890560989306495
7.3890560989306495
7.38905609893065
0.0
1e-25
```

对数函数 log(x,base)用于计算 $\log_{base}x$。此外，该函数还有一些变体，如 log2(x)、log10(x)、log1p(x)分别用于计算 $\log_2(x)$、lg(x)、lg(1+x)。例如：

```
x = 9
print(math.log(x))     # 自然对数
print(math.log(x,3))   # 以 3 为底的对数
#底是 10 的对数,log10(x)比 log(x,10)精度更高
print('{:10.8f} {:20.18f}'.format(math.log10(x),math.log(x,10)))
#底是 2 的对数,log2(x)比 log(x,2)精度更高
print('{:10.8f} {:20.18f}'.format(math.log2(x),math.log(x,2)))
#对于接近于 0 的 x，log1p(x)比 lg(1+x)更准确
xx = 0.00000000000000000000000001
print(math.log(1 + xx),math.log1p(xx))
```

输出：

```
2.1972245773362196
0.95424251 0.954242509439324871
3.16992500 3.169925001442312595
0.0 1e-25
```

6. 三角函数

(1)角度、弧度转换。

函数 radians()将角度值转换成弧度。例如，rad = radians(deg)将角度值 deg 转换为弧度值 rad，相当于 rad = deg * π / 180。函数 degrees()将弧度值转换成角度值。例如，deg = degrees(rad) 将弧度值 rad 转换成角度值 deg，相当于 deg = rad * 180 / π。

例如：

```
print(math.radians(45))            #角度值转换为弧度值
print(math.degrees(math.pi/2))     #弧度值转换为角度值
```

输出：

```
0.7853981633974483
90.0
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

(2)三角函数及反三角函数。

math 模块包含了各种三角函数和反三角函数，如 sin()、cos()、tan()、atan()等函数；还有其他函数，如双曲函数、特殊函数（如正态分布函数）等。例如：

```
print(math.sin(math.pi/2))          #计算正弦函数
print(math.tan(math.pi/4))          #计算正切函数
print(math.degrees(math.atan(1)))   #计算反正切函数，将结果由弧度值转换为角度值
print(math.factorial(5))            #阶乘  5! = 1×2×3×4×5
```

输出：

```
1.0
0.9999999999999999
45.0
120
```

更多 math 模块中的数学函数，读者可查看网址：https://docs.python.org/3/library/math.html。

总结

- Python 的数值类型有 int、float、complex、bool。
- 通过在数值前面加前缀可表示不同进制的数值。内置函数 bin()、oct()、hex()可分别用于得到二进制、八进制、十六进制数值对应的字符串。
- 不同类型的数值参与运算时，有时会自动进行隐式类型转换，有时则需要进行显式强制类型转换。内置函数 int()、float()、complex()将其他类型的值，如字符串类型 str 的值强制显式类型转换为 int、float、complex 类型的值。内置函数 str()可将其他类型的值转化为字符串类型。
- 浮点数只能用二进制近似表示。Python 提供了用于分数计算的 Fraction 类型。
- math 模块、cmath 模块的函数可以进行各种数值计算。

4.2 列表

4.2.1 列表的定义

Python 提供了一些组合数据类型，用于将多个值组合在一起，其中，最常用的组合数据类型就是列表(list)。通过将所有值写在一对方括号[ ]里并以逗号，隔开就可以创建一个 list 对象。列表中的值(对象)称为列表中的数据元素。这些元素的数据类型可以不同。例如：

```
[1, "Hello", 3.4]
```

定义了一个包含三个元素的 list 对象，这三个元素的类型分别是 int、str 和 float 类型。

为了能重复对同一个 list 对象进行操作，可以用变量名引用这个 list 对象。例如：

```
my_list = [1, "Hello", 3.4]
print(my_list)
type(my_list)
```

输出：

```
[1, 'Hello', 3.4]
list
```

既然 list 可以包含不同类型的元素，list 里当然还可以包含其他的 list。例如：

```
my_list = [2,  [8, 4, 6],'小白',3.14, 'python',  ['a']]
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

可以用 Python 的内置函数 len() 查询一个 list 中的元素个数。例如：

```
len(my_list)
```

输出：

```
6
```

4.2.2 访问 list 的元素（索引和切片）

和字符串一样，list 是一个有序集合，每个元素都有一个确定的下标，下标从 0 开始，长度为 *n* 的列表的最后一个元素的下标是 *n*–1。可以用**下标运算符[]**访问 list 的元素。例如：

```
my_list = [2,  [8, 4, 6],'小白',3.14, 'python',  ['a']]
print(my_list)
print(my_list[0])     #第 1 个元素的下标是 0
print(my_list[2])
print(my_list[1][2])  #下标为 1 的元素本身也是一个 list，可以对它继续用下标访问其元素
```

输出：

```
[2, [8, 4, 6], '小白', 3.14, 'python', ['a']]
2
小白
6
```


下标也可以是负数，最后一个元素的下标是–1,倒数第 2 个元素的下标就是–2，……例如：

```
print(my_list[-1])
print(my_list[-2])
```

输出：

```
['a']
python
```

下标不能超出范围，如正的下标的范围为 `[0,len(my_list)-1]`，而负的下标的范围为 `[-len(my_list), -1]`。例如：

```
print(my_list[5])     #最后一个元素,等价于 print(my_list[ len(my_list)-1]
print(my_list[ len(my_list)-1])

print(my_list[-6])    #第 1 个元素,等价于 print(my_list[ -len(my_list)]
print(my_list[-len(my_list)])
```

输出：

```
['a']
['a']
2
2
```

注意，下标不能超出范围。例如：

```
print(my_list[6])
print(my_list[-7])
```

抛出异常：

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-15-6c007bb3fdc9> in <module>()
      1 #下标超出范围
----> 2 print(my_list[6])
      3 print(my_list[-7])
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

对于 my_list=[2 , 3.14 , [8,11] , 'python' , 9 , 'hello']，其每个元素的正负下标如表 4-1 所示。

表 4-1　list 的元素的正负下标

my_kust	2	3.14	[8,11]	'python'	9	'hello'
正下标	0	1	2	3	4	5
负下标	–6	–5	–4	–3	–2	–1

可以用**切片操作(slicing operator)**访问一个范围内的多个元素，其格式为：

```
list[start:end:step=1]
```

表示查询从 start 到 end 之间(但不包含 end)间隔步长为 step 的元素。如果没有指定 step，则表示 step =1，即间隔步长为 1；如果没有指定 start，则表示从第 1 个元素(start=0)开始；如果没有指定 end，则表示到最后 1 个元素。例如：

```
my_list = [ 2 , 3.14 , [8,11] , 'python' , 9 , 'hello']
print(my_list[2:5])   #从第 3 个到第 6 个之间(下标[2:5])，不包含第 6 个
print(my_list[:-5])   #从第 1 个到倒数第 5 个，不包含倒数第 5 个
print(my_list[5:])    #从第 6 个往后的所有元素
print(my_list[1:5:2]) #第 2 个到第 6 个元素之间(下标[1:5])，间隔步长为 2
print(my_list[::2])   #间隔步长为 2
print(my_list[:])     #所有元素
```

输出：

```
[[8, 11], 'python', 9]
[2]
['hello']
[3.14, 'python']
[2, [8, 11], 9]
[2, 3.14, [8, 11], 'python', 9, 'hello']
```

list 是**可变的(mutable)**，可以通过**下标(切片)**修改其中的 1 个或多个元素。对切片的修改还有“插入”或“删除”的效果。例如：

```
my_list=[2, 3.14, [8,11]. 'Python', q. 'hello']
my_list[1] = 34           #下标为 1 的元素修改为 34
print(my_list)
my_list[2:5] = [25,8]     #用 1 个新的 list[25,8]替换切片 my_list[2:5]
print(my_list)
my_list[2:5]=[]           #用 1 个空的 list 替换切片，相当于删除切片中的元素
print(my_list)
my_list[2:2]=["hello","world"]     #相当于在下标为 2 的元素处插入一组元素
print(my_list)
```

输出：

```
[2, 34, [8, 11], 'python', 9, 'hello']
[2, 34, 25, 8, 'hello']
[2, 34]
[2, 34, 'hello', 'world']
```

4.2.3　包含和遍历

用 in 可判断 1 个元素是否在 1 个 list 里。例如：

```
my_list = ['p','r','o','b','l','e','m']
print('p' in my_list)         #输出: True
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print('a' in my_list)          #输出: Fasle
print('c' not in my_list)      #输出: True
```

输出：

```
True
False
True
```

用 for 迭代遍历 1 个 list。例如：

```
for fruit in ['apple','banana','mango']:
   print("I like",fruit)
```

输出：

```
I like apple
I like banana
I like mango
```

4.2.4 list 的算术运算

可以通过加法运算+拼接两个 list，用整数和 list 的乘法运算*复制 list 内容。例如：

```
odd = [1, 3, 5]
print(odd + [9, 7, 5])    #输出:[1, 3, 5, 9, 7, 5]
print(["re"] * 3)         #输出:["re", "re", "re"]
```

输出：

```
[1, 3, 5, 9, 7, 5]
['re', 're', 're']
```

4.2.5 Python 的内置函数对 list 进行操作

可以用 Python 的内置函数对 list 进行操作，如函数 del()可删除 list 的中 1 个元素或 1 个范围里的元素(切片)。例如：

```
alist = ['p','r','o','b','l','e','m']
a = alist
del alist[2]      #删除 1 个元素
print(alist)      #输出:  ['p', 'r', 'b', 'l', 'e', 'm']
del alist[1:5]    #删除多个元素
print(alist)      #输出: ['p', 'm']
del alist         #删除整个  list
```

输出：

```
['p', 'r', 'b', 'l', 'e', 'm']
['p', 'm']
```

当 del 作用于指向 list 的变量本身时，这个变量名从系统中被删除，它的 list 的引用计数减 1，只有当 list 的引用计数为 0 时，这个 list 才被删除。

```
del alist         #删除 alist 变量，但 alist 指向的 list 引用计数仍不为 0
print(a)          #因为变量 a 仍引用这个对象
print(alist)      #访问 1 个不存在的变量，是错误的
```

最后一句访问 1 个已经删除的变量 alist，是错误的：

```
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    print(alist)
NameError: name 'alist' is not defined
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

1．枚举函数 enumerate()

对 1 个列表，当既要访问索引又要遍历元素时，可以这样写：

```
alist = [3,7,'hello']
for i in range (len(alist)):
   print(i ,alist[i])
```

输出：

```
0 3
1 7
2 hello
```

更好的办法是使用枚举函数 enumerate()。对于一个可迭代的(iterable)的对象(如列表、字符串)，枚举函数 enumerate()将其组成一个索引序列，即一个“索引和值”的序列。例如：

```
for i,value in enumerate(alist):
     print(i ,value)
```

输出：

```
0 3
1 7
2 hello
```


枚举函数 enumerate()还可以接收第二个参数，用于指定索引起始值，如：

```
for i,value in enumerate(alist):
   print(i ,value)
```

输出：

```
1 3
2 7
3 hello
```

2．sorted()

内置函数 sorted()可以用来对一个 list 排序，但它不修改原来的 list 而是返回一个排好序的新的 list。格式为：

```
sorted(iterable, key=None, reverse=False)
```

其中，iterable 是可迭代对象，key 是用来进行比较的函数，reverse 表示是“逆序”(True)还是“正序”(False)(默认)排列。例如：

```
alist = [19,2,17,12]
blist = sorted(alist)
print(blist)
blist = sorted(alist,reverse = True)
print(blist)
alist = [('b',2),('a',1),('c',3),('d',4)]
blist = sorted(alist, key=lambda x:x[1])       #利用 key
print(blist)
```

输出：

```
[2, 12, 17, 19]
[19, 17, 12, 2]
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

3．可处理 list 的内置函数

以下为一些可处理 list 的常用函数。

- all()　当 list 为空或所有元素为 True 时返回 True，否则返回 False。
- any()　当 list 不为空且有一个元素为 True 时返回 True，否则返回 False。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- enumerate()　返回一个 enumerate 对象，包含了所有元素的索引和值 tuple。
- len()　返回 list 的长度(数据元素的个数)。
- list()　将一个可迭代对象(tuple、set、dict)转换为一个 list。
- max()　返回 list 中的最大值元素。
- min()　返回 list 中的最小值元素。
- sorted()　返回新的排好序的 list(原来的 list 不改变)。
- sum()　返回 list 中所有元素的和。

4.2.6　list 的方法

Python 中的一切都是对象，对象总是某个类(class)的一个具体实例，每个类都描述了该类的所有对象的共同属性，这些属性除表示对象状态的数据外，还包括对类对象进行操作处理的函数。这些函数不同于前面介绍的全局函数，它们属于这个类内部的函数，也称**类的成员函数或方法**。

list 作为一个类，也有很多的方法(成员函数)，这些方法可以对一个 list 进行**追加、插入、删除**等操作。例如，list 的 **append()** 和 **extend()** 方法分别可以在一个 list 最后添加一个元素或多个元素。

一个类的方法必须通过一个具体的类对象才能调用。例如，list 的 append()方法必须作用于一个具体类对象，这样才能对这个对象进行操作。例如：

```
odd = [1, 3, 5]
odd.append(7)              #最后添加一个元素
print(odd)                 #输出: [1, 3, 5, 7]

odd.extend([9, 11, 13])    #扩展原来的 list
print(odd)                 #输出: [1, 3, 5, 7, 9, 11, 13]
```

输出：

```
[1, 3, 5, 7]
[1, 3, 5, 7, 9, 11, 13]
```

list 的 insert()方法可以在中间某个位置插入一个元素。例如：

```
odd = [1, 9]
odd.insert(1,3)            #在下标为 1 的元素处插入一个整数 3
print(odd)                 #输出: [1, 3, 9]

odd.insert(1,[4,5])        #在下标为 1 的元素处插入一个 list, [4,5]
print(odd)
```

输出：

```
[1, 3, 9]
[1, [4, 5], 3, 9]
```

list 的 pop()方法可以删除指定下标的元素，如果没有指定下标，则删除最后一个元素。而 list 的 clear()方法则清空整个 list，但没有销毁它。例如：

```
my_list = ['p','r','o','b','l','e','m']
my_list.pop(2)
print(my_list)
my_list.pop()
print(my_list)
my_list.clear()
print(my_list)
```

输出：

```
['p', 'r', 'b', 'l', 'e', 'm']
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
['p', 'r', 'b', 'l', 'e']
[]
```

list 的 remove()方法可以删除特定的元素。注意，此时传递给 remove()方法的是这个元素的值而不是下标。

```
my_list = ['p','r','o','b','l','e','m']
my_list.remove('p')
print(my_list)        # 输出: ['r', 'o', 'b', 'l', 'e', 'm']
```

输出：

```
['r', 'o', 'b', 'l', 'e', 'm']
```

list 的 sort()方法可以对 list 进行排序，格式是：

```
sort(key=None, reverse=False)
```

key 是用来进行比较的函数，reverse 表示是“逆序”还是“正序”排列，默认是“正序”排列。例如：

```
vowels = ['e', 'a', 'u', 'o', 'i']
vowels.sort()  #以正序排序
print('正序:', vowels)
vowels.sort(reverse=True)   # 以逆序排序
print('逆序:', vowels)
```


输出：

```
正序: ['a', 'e', 'i', 'o', 'u']
逆序: ['u', 'o', 'i', 'e', 'a']
```

通过指定函数 key，可以指定排序时元素的关键字值。例如：

```
# 获取 elem 的第 2 个元素作为 key 的值
def takeSecond(elem):
    return elem[1]
pairs = [(2, 'two'), (1, 'one'), (3, 'three'), (4, 'four')]
pairs.sort(key=takeSecond)
print(pairs)
```

输出：

```
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

也可以用函数 lambda()代替普通的函数。例如：

```
pairs = [(2, 'two'), (1, 'one'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[0])
print(pairs)
```

输出：

```
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

sort()方法和函数 **sorted**()的区别是：list 的 sort()方法会修改它作用的 list，而函数 sorted()不会修改原 list，而是返回一个新的 list。

下面是一些 list 的常用方法，感兴趣的读者可以参考相关文档或通过帮助函数 help()学习如何使用这些方法对 list 进行操作。

- append()　将一个元素加到 list 的最后。
- extend()　将一个 list 中所有元素加到另外一个 list 的后面。
- insert()　在指定下标(索引)元素处插入一个元素。
- remove()　从 list 里删除一个元素。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- pop()　删除并返回指定索引位置的元素。
- clear()　删除 list 里的所有元素。
- index()　返回匹配元素的索引。
- count()　返回元素在 list 里出现的次数。
- sort()　以递增方式对 list 元素排序。
- reverse()　将 list 里的元素逆序。
- copy()　返回 list 的一个浅拷贝。

4.2.7　列表解析式

在创建一个 list 时，除在[]中直接给出所有列表元素外，还可以通过一个 for 循环语句生成列表的所有元素。这种创建 list 的方式称为**列表解析式**(list comprehension)。

例如：

```
pow2 = [2 ** x for x in range(10)]
# 输出: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
print(pow2)
```

输出：

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```


注意，“[2 ** x for x in range(10)]”等价于：

```
pow2 = []
for x in range(10):
   pow2.append(2 ** x)
print(pow2)
```

输出：

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

4.2.8　list 包含的不是对象本身而是对象的引用

list 中的元素不是对象本身而是对象的引用。无论每个对象本身占据多大内存，这个引用总是占用固定大小的内存。例如：

```
import sys

alist = []
print(sys.getsizeof(alist) ,end = ' ')   #输出 64
alist = [2]
print(sys.getsizeof(alist) ,end = ' ')   #输出 72
alist = [2,3.14]
print(sys.getsizeof(alist) ,end = ' ')   #输出 80
alist = [2,3.14,'hello']
print(sys.getsizeof(alist) ,end = ' ')   #输出 88
alist = [2,3.14,'hello',[5,6,'world']]
print(sys.getsizeof(alist) ,end = ' ')   #输出 96
```

输出：

```
64 72 80 88 96
```

空的 list 占用 64 字节的内存，每添加一个元素，就多占用 8 字节，这 8 字节就是实际对象的引用所消耗的内存。如图 4-1 所示，alist 的内存布局示意图。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

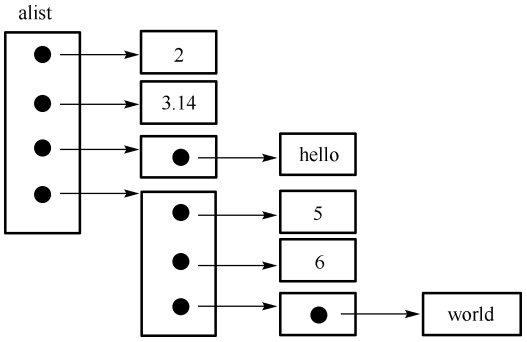


图 4-1　alist 的内存布局示意图

总结

- 创建 list 列表对象有三种方法：方括号[]指明所有元素、内置函数 list()、列表解析式将可迭代对象转为一个 list，生成所有元素。
- list 是有序序列，每个元素都对应唯一的下标，可以通过下标运算符[]进行索引或切片操作，也可以用 for 循环迭代访问其中的元素。
- list 中的每个元素都是实际对象的引用，而不是实际对象本身。
- 既可以用内置函数对 list 进行操作，也可以用 list 的方法对 list 进行操作。

4.3　字符串

4.3.1　定义字符串

1．字符串(str)

Python 没有表示单个字符的字符类型，只有字符串类型 str。字符串是用单引号或双引号括起来的一系列字符。

单引号表示的字符串中可以包含双引号字符，但不能直接包含单引号字符(否则无法知道字符串的开始和结尾分别在哪里)。同样，双引号表示的字符串中可以包含单引号字符，但不能直接包含双引号字符。例如：

```
print("https://a.hwdong.com")
print("教'小白'精通编程")
print('教"小白"精通编程')
print('a')
print('教')
print(type("教'小白'精通编程"))
print(type('教"小白"精通编程'))
print( type('a') )
print( type("教") )
```

输出：

```
https://a.hwdong.com
教'小白'精通编程
教"小白"精通编程
a
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
教
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

双引号字符串中不能直接包含其他双引号，单引号字符串不能直接包含其他单引号。例如：

```
print("Hello " world")  #: 错：双引号字符串中包含了其他双引号
print('Hello ' world')  #: 错：单引号字符串中包含了其他单引号
```

输出：

```
  File "<ipython-input-2-d83125c2b08f>", line 1
    print("Hello " world")  #: 错：双引号字符串中包含了其他双引号
                      ^
SyntaxError: invalid syntax
```

前面讲过，字符可以有不同的编码标准，Python3 字符采用 **UniCode** 编码点表示，而编码方式默认采用 UTF-8 编码方式(用 UTF-8 编码方式表示字符的 Unicode 编码点)。例如：

```
import sys
sys.getdefaultencoding()
```

输出：

```
'utf-8'
```


对于一个字符串，可以用函数 **len**()得到其长度(字符串中字符的个数)。

```
print(len("教"),end = ' ' )
print(len("a"),end = ' ' )
print(len("教小白精通编程"),end = ' ' )
```

输出：

```
1 1 7
```

2．作用于字符串的运算符+、*

可以用+拼接两个字符串，可以用*重复一个字符串。例如：

```
# 3 乘'un'，表示重复 3 个'un',然后加上一个'ium'
3 * 'un' + 'ium'
```

输出：

```
'unununium'
```

当然，字符串文字量可以通过紧挨着写，达到拼接的效果。例如：

```
s = 'Py' 'thon'
print(s)
```

输出：

```
Python
```

如果两个需要拼接的字符串文字量不在同一行，则需要在前面的字符串后面加上反斜杠字符\，或者在两个字符串外面用左右圆括号包围它们。例如：

```
text = 'Put several strings within parentheses '\
       'to have them joined together.'
print(text)
text = ('Put several strings within parentheses '
       'to have them joined together.')
print(text)
```

输出：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        Put several strings within parentheses to have them joined together.
        Put several strings within parentheses to have them joined together.
```

但是，字符串变量和字符串文字量不能这样拼接，即不能通过上述方式(紧挨着写、反斜杠、圆括号)达到拼接效果。必须使用+运算符。例如：

```
        prefix ='Py'
        prefix 'thon'
```

第二句代码产生“invalid syntax(无效语法)”的语法错误：

```
          File "<ipython-input-10-95a06b9fd091>", line 1
            prefix 'Py'
                   ^
        SyntaxError: invalid syntax
```

应使用+运算符：

```
        prefix = 'Py'
        prefix + 'thon'
```

输出：

```
        'Python'
```


4.3.2 转义字符

1. 转义字符

如要在单引号表示的字符串中包含单引号，则可以在其前面添加斜杠符号\构成一个单独的字符\'表示单引号字符。这种前面加斜杠字符表示的字符\'称为**转义字符**，它表示的是一个字符而不是两个字符。

```
        print('hello \'li ping\'')
        print(len('hello \'li ping\''))   #打印字符串的长度, \'表示的是一个字符
```

输出：

```
        hello 'li ping'
        15
```

如何表示反斜杠字符呢？办法是在它前面同样加上反斜杠字符，即转义字符\\表示的是单个反斜杠字符\。

```
        print('hello \\')
```

输出：

```
        hello \
```

转义字符常用于表示各种不可见的字符，如控制字符，如\t 表示“制表符”，\n 表示“换行符”。例如：

```
        print('hello\tworld\n\n')
        print('haha')
```

输出：

```
        Hello world


        haha
```

print()遇到制表符\t 时，输出了几个空格，而遇到\n 时，则输出换了一行。

使用三个双引号或单引号("""...""" 或'''...''')可以表示多行字符串。Python 会自动在每行的结束处添加一个“行结束符”，即\r，也可以在该行后面添加\阻止这样做。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print("""Usage:\
thingy [OPTIONS]
    -h                        Display this usage message
    -H hostname               Hostname to connect to
""")
```

输出：

```
Usage: thingy [OPTIONS]
    -h                        Display this usage message
    -H hostname               Hostname to connect to
```

注意

反斜杠和其后的字符如果不构造转义字符，则表示的是两个普通字符，如"\p"表示的是两个普通字符\和 p。例如:

```
print("he\p")
```

输出:

```
he\p
```

2．原始字符串(raw)

有时可能希望忽略字符串中的转义字符，则可以将 r 或 R 放在字符串前面，表示它是一个原始字符串，其中的任何转义字符都将被忽略。例如：

```
print(r'hello\t\tworld\n')   # \t 和\n 是两个字符而不是一个转义字符
print('hello\t\tworld\n')
```

输出：

```
hello\t\tworld\n
hello  world
```

4.3.3 索引

字符串中的每个字符都有一个确定的下标，第 1 个字符的下标是 0，第 2 个字符的下标是 1，…，依此类推。可以通过下标运算符[]访问字符串中的字符。

```
word = 'Python'
print(word[0])
print(word[5])
print(type(word[0]))   #一个字符也是一个字符串
```

输出：

```
P
n
<class 'str'>
```

但是下标不能超出范围，如长度为 *n* 的最后一个字符的下标是 *n*–1，如果用超出 *n*–1 的下标就会出错。例如：

```
print(word[6])
```

抛出异常：

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-7c3e69df5e88> in <module>()
----> 1 print(word[6])

IndexError: string index out of range
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

字符串的下标也可以是负整数，其中最后一个字符的下标是–1，倒数第二个字符的下标是–2，…，依此类推。

```
print(word[-1],end = ' ')
print(word[-2],end = ' ')
print(word[-6],end = ' ')
```

输出：

```
n o P
```

字符串 python 的每个字符对应的下标如表 4-2 所示。

表 4-2　字符串 python 的每个字符对应的下标

p	y	t	h	o	n
0	1	2	3	4	5
–6	–5	–4	–3	–2	–1

4.3.4　切片

和 list 一样，不仅可以用下标访问单个字符，还可以用两个下标(起始下标和结束下标)访问这两个下标范围里的一系列字符，并返回这些字符构造的一个字符串。这种通过指定下标起始和结束位置访问字符串中一系列字符的方式称为**切片(slicing)**。

假设 str 是一个字符串，切片操作格式如下：

```
str[start:end:step]
```

返回的是一个包含 start 位置字符但不包含 end 位置的，步长间隔是 step 的那些元素构成的一个字符串。start 默认为 0，end 默认是最后一个字符的下一个位置，step 默认是 1。

```
print(word[0:2])
print(word[2:5])
print(word[:2])
print(word[2:])
print(word[:-2])
print(word[-3:])
print(word[:])  #表示获取整个字符串
```

输出：

```
Py
tho
Py
thon
Pyth
hon
Python
```

4.3.5　字符串不可修改

list 对象是可以修改的，即可以通过下标(切片)，或者 list 的方法或内置函数修改一个 list 对象的内容。而 str 字符串对象是不可变的，即**不可修改**的。例如：

```
s = 'hello'
s[2] ='A'
```

抛出错误异常：

```
---------------------------------------------------------------------------
TypeError            Traceback (most recent call last)
<ipython-input-27-1ce2af98efe6> in <module>()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    1 s = 'hello'
----> 2 s[2] ='A'

TypeError: 'str' object does not support item assignment
```

但字符串对象是可以用 del 从内存中删除的。例如：

```
del s
```

只能删除整个字符串对象，不能删除字符串中的某个字符，因为字符串是不可修改的。例如：

```
s = 'hello'
del s[2]
```

4.3.6 包含和遍历

和 list 一样，可以用 in 判断一个字符是否在一个字符串对象中。例如：

```
s= 'hello'
print('h' in s)
print('H' in s)
```

输出：

```
True
False
```

和 list 一样，可以用 for 遍历(迭代访问)一个字符串中的每个字符。例如：

```
s= 'hello, world'
count = 0
for c in s:
   if(c == 'l'):
      count += 1
print(count)
```

输出：

```
3
```

4.3.7 内置函数对字符串操作

枚举函数 enumerate()返回一个包含所有元素的(下标、值)的 enumerate 对象，可以将 enumerate 对象看作类似 list 对象的一个列表。例如：

```
str = 'hello'
list_enumerate = list(enumerate(str))
print(enumerate(str))
print(list_enumerate)
for index,value in enumerate(str):
  print(index,value)
```

输出：

```
<enumerate object at 0x000001B9A1E291B0>
[(0, 'h'), (1, 'e'), (2, 'l'), (3, 'l'), (4, 'o')]
0 h
1 e
2 l
3 l
4 o
```

函数 **len**()返回一个字符串中的字符个数。例如：

```
print(len('hello'))
print(len('hello\tworld'))  #\t 是转义字符，因此是一个字符，所以输出应该是 11
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
5
11
```

4.3.8 字符串的方法

前面看到字符串可以用函数，如 len()得到它的长度，可以用函数 print()输出字符串内容。另外，字符串作为一个数据类型，它还有属于字符串类型自身的函数(称为方法)，这些方法必须通过一个具体的字符串对象来调用。

例如，字符串有一个叫作 upper()的方法，该方法返回一个将字符串中的字母转为大写字母的新的字符串。例如：

```
s = 'hello'
ret = s.upper()
print(s,ret)
```

输出：

```
hello HELLO
```

还可用 lower()方法返回一个将字符串中的字母转为小写字母的新字符串。

1．删除空白字符(lstrip()、rstrip()、strip())

lstrip()、rstrip()和 strip()分别用于删除字符串中的左侧、右侧和左右两侧位置的空白字符。例如：

```
s2 = '   hello,  world   '
print('#', s2.lstrip(),'#')
print('#', s2.rstrip(),'#')
print('#', s2.strip(),'#')
```

输出：

```
# hello,  world    #
#    hello,  world #
# hello,  world #
```

2．分割(split())

split() 通过指定分隔符对字符串进行分割，其语法格式如下：

```
split(sep=None, max=-1)
```

sep 是分隔符，默认为空格，max 表示最大分割次数，可产生 max+1 个分割的字符串。max 的默认值是–1，表示分割次数没有限制。该方法返回分割后的字符串列表。例如：

```
s = 'hello world'
print(s.split())
s = 'hello,world'
print(s.split())
print(s.split(','))  #指定分隔符好
s = r'https:\\a.hwdong.com'
print((s.split('.',1)) )
```

输出：

```
['hello', 'world']
['hello,world']
['hello', 'world']
['http:\\a', 'hwdong.com']
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

3．连接（join()）

join()方法的语法格式如下：

```
str.join(sequence)
```

即用这个字符串 str 连接序列 sequence 中的其他字符串。该方法返回连接后的新字符串。例如：

```
availability = ["Monday", "Wednesday", "Friday", "Saturday"]
result = " - ".join(availability)  #用" - "连接 availability 中的字符串
print(result)
```

输出：

```
Monday - Wednesday - Friday - Saturday
```

4．替换（replace()）

replace()方法的语法格式如下：

```
replace(old, new [, max])
```

将字符串中匹配 old 的子串替换为字符串 new，如果指定 max，则最多替换 max 次。该方法返回一个新的字符串。例如：

```
string = 'hello world, hello world, hello world'
string.replace('wor','woo',2)
```

输出：

```
'hello woold, hello woold, hello world'
```

5．查找（find()和 index()）

find()和 index()方法的语法格式如下：

```
find(str, beg=0 end=len(string))
index(str, beg=0, end=len(string))
```

这两个方法都是在字符串中查找是否存在匹配字符串的 str 子串。其中，beg 和 end 分别表示查找范围的开始和结束位置。如果查找成功，则返回匹配字符串子串的位置；如果失败，则 find()方法返回–1，而 index()方法会抛出一个 value error 的错误异常。

另外，还有两个对应的方法用于从字符串结尾反方向查询是否存在匹配的子串，分别是：

```
rfind(str, beg=0 end=len(string))
rindex(str, beg=0, end=len(string))
```

例如：

```
string = 'hello world, hello world'

result = string.index('world')
print("子串索引:", result)
result = string.rindex('world')
print("子串索引:", result)
```

输出：

```
子串索引: 6
子串索引: 19
```

find()只能用于字符串，而 index()也是 list.tuple 类型的方法。

6．子串的计数（count()）

count()方法的语法格式如下：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
count(str, beg= 0,end=len(string))
```

这个方法用于计算 beg 和 end 范围里出现 str 子串的次数。例如：

```
s3= 'hello world, hello world,hello world'
print(s3.count('wor'))
print(s3.count('wor',8))  #从第 9 个字符后开始统计
```

输出：

```
3
2
```

7．格式化 (format())

下面的拼接字符串不是很符合 Python 的风格：

```
name = "LiPing"
where= 'shanghai'
work_in = name + "在" + where+"工作！"
print(work_in)
```

输出：

```
LiPing 在 shanghai 工作！
```


Python 的 str 类型有一个更好的字符串“格式化”方法 format()。例如：

```
work_in = "{} 在{} 工作！".format(name, where)
print(work_in)
```

输出：

```
LiPing 在 shanghai 工作！
```

format()方法针对一个包含占位符{}的字符串，将参数传给 format()方法来取代占位符{}。format()方法不仅可以接收字符串，还可以接收其他类型数据。例如：

```
age = 28
work_in = "{} 在{} 工作了{}年".format(name, where, age)
print(work_in)
```

输出：

```
LiPing 在 shanghai 工作了 28 年
```

和普通函数一样，format()方法可以接收默认位置参数、位置参数、关键字参数等。占位符{}里可以指定参数的位置、名字。例如：

```
#默认位置参数：依次用实参取代{}占位符
print("Hello {}, your balance is {}.".format("Li", 70.5))
#位置参数
print("Hello {0}, your balance is {1}.".format("Li", 70.5))
#关键字参数
print("Hello {name}, your balance is {score}.".format(name="Li", score=70.5))
#混合参数
print("Hello {0}, your balance is {score}.".format("Li", score=70.5))
```

输出：

```
Hello Li, your balance is 70.5.
Hello Li, your balance is 70.5.
Hello Li, your balance is 70.5.
Hello Li, your balance is 70.5.
```

数值的格式化以：开头，后面跟一些字符，如:d 表示输出十进制的整数，:f 表示输出浮点

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

数，:b、:o、:x 分别表示输出的是二进制、八进制和十六进制数。例如：

```
print("The number is:{:d}".format(123))
print("The float number is:{:f}".format(123.4567898))
print("bin: {0:b}, oct: {0:o}, hex: {0:x}".format(12))
```

输出：

```
The number is:123
The float number is:123.456790
bin: 1100, oct: 14, hex: c
```

还可以用于指定数值的宽度并填充。例如：

```
#指定整数的最小宽度
print("{:5d}".format(12))
#对超出宽度的整数不起作用
print("{:2d}".format(1234))
#浮点数的宽度、精度
print("{:8.3f}".format(12.23))
#整数的最小宽度、填充 0
print("{:05d}".format(12))
#浮点数填充 0
print("{:08.3f}".format(12.2346))
```


输出：

```
   12
1234
  12.230
00012
0012.235
```

还可以用于指定对齐方式，<、>、^分别表示左对齐、右对齐、中间对齐，而=表示强制正、负号(+、–)位于最左边。例如：

```
# 整数右对齐
print("{:5d}".format(12))
# 浮点数中间对齐
print("{:^10.3f}".format(12.2345))
# 整数左对齐，填充 0
print("{:<05d}".format(12))
#浮点数中间对齐
print("{:=8.3f}".format(-12.23))
#中间对齐，用*填充
print("{:*^8}".format("cat"))
```

输出：

```
   12
   12.235
   12000
- 12.230
**cat***
```

format()方法还可以接收 dict(字典)实参，此时需要用“解封参数列表”方式将 dict 中的每一项分离并传递给该方法。例如：

```
person = {'age': 23, 'name': 'LiPing'}
print("{name}'s age is: {age}".format(**person))
```

输出：

```
LiPing's age is: 23
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

8．str()和 repr()

每个对象都有两个方法：__str__()和__repr__()分别返回适合阅读的字符串和解释器内部表示的字符串。例如：

```
s = 'hello'
s.__str__()
```

输出：

```
'hello'
```

执行：

```
s.__repr__()
```

输出：

```
"'hello'"
```

用 print()函数输出__str__()、__repr__()的值：

```
print(s.__str__())
print(s.__repr__())
```

输出：

```
hello
'hello'
```


二者的区别是，__str__()返回的字符串没有引号，而__repr__()在机器内部表示字符串是有引号的。

在 fromat()方法中，str()和 repr()的快捷键是!s 和!r。例如：

```
"{0!r}和{0!s}".format('hello')  # 0 对应的是第 1 个参数
```

输出：

```
"'hello'和 hello"
```

总结

- 字符串是有序的字符序列，每个字符都对应唯一的下标，可以通过下标运算符[]进行索引或切片操作，也可以用 for 循环迭代访问其中的字符。
- 字符串可以用单引号、双引号或三个引号表示，特殊字符需要用转义字符，如\r、\n、\t、\"、\\，Python 中用 raw 字符串更简单。
- 字符串支持 in、+、*等运算符。
- 字符串是不可变的对象。
- 可以用内置函数或字符串 str 类型的方法对字符串进行操作。例如，str 的 format()方法可以对字符串进行各种格式化操作，产生一个新的字符串。

4.4 元组

4.4.1 创建 tuple 对象

1．元组(tuple)

元组(tuple)是和 list 类似的一个有序序列，但 tuple 是不可变(不可修改)的，而 list 对象是可修改的(添加、删除、修改)。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

创建一个元组(tuple)对象的方法是，将所有值用逗号隔开并放在一对圆括号里。例如：

```
t = ("alpha", "delta", "omega")
print(t)
```

输出：

```
('alpha', 'delta', 'omega')
```

和 list 一样，tuple 对象的元素类型可以不同，可以包含 list、tuple 等各种类型的对象。例如：

```
t = ("alpha",2,3.14,[5,'hello'], ("delta", "omega"))
```

创建只包含一个元素的 tuple 时，必须在这个元素后面加一个逗号,，否则就不是一个 tuple。例如：

```
t7 = (3,)   # 这是一个 tuple
t8 = (3)   # 这不是一个 tuple
print(type(t7))
print(type(t8))
```

输出：

```
<class 'tuple'>
<class 'int'>
```

甚至可以不用左右圆括号就能创建一个 tuple。例如：

```
t9 = 2,'hello',3.14
print(type(t9))
```

输出：

```
<class 'tuple'>
```

2. 用函数 tuple()创建 tuple 对象

该函数可以接收任何序列类型或可迭代对象(关于“可迭代对象”，将在本书第 5 章介绍)，用于创建一个元组(tuple)。

下面代码将字符串、list 对象、tuple 对象等传给函数 tuple()，创建一个 tuple 对象：

```
t1 = tuple('hello')
print(t1)
t2 = tuple([2,5,8])       #接收一个 list 对象
print(t2)
t3 = tuple((3,6,9))       #接收一个 tuple 对象
print(t3)
t4 = tuple([x * 2 for x in range(1, 10)])  #接收“列表解析式”产生的 list 对象
print(t4)
t5 = tuple(range(1, 10))  #range 函数产生一个可迭代对象
                          #其值是 1 到 10 之间的整数(不包含 10)
print(t5)
t6 = tuple(i for i in (1,2,3))     #tuple()可接收一个解析式
print(t6)
t7 = (i for i in (1,2,3)) #产生的是一个生成器 generator 对象，而不是一个 tuple 对象
print(t7)
```

输出：

```
('h', 'e', 'l', 'l', 'o')
(2, 5, 8)
(3, 6, 9)
(2, 4, 6, 8, 10, 12, 14, 16, 18)
(1, 2, 3, 4, 5, 6, 7, 8, 9)
(1, 2, 3)
<generator object <genexpr> at 0x0000023733195BF8>
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

3．tuple 解封(tuple unpacking)

tuple 可以一次同时为多个变量赋值，这称为 **tuple 解封**。

```
x,y,z = (2,'hello',3.14)
print(x,end=' ')
print(y,end=' ')
print(z,end=' ')
```

输出：

```
2 hello 3.14
```

因为创建 tuple 时圆括号是可选的，所以也可以写为如下形式：

```
x,y,z = 2,'hello',3.14
print(x,end=' ')
print(y,end=' ')
print(z,end=' ')
```

左边的变量个数和右边的 tuple 中的元素个数必须一样。例如：

```
x,y,z = 2,'hello'
```

产生“not enough values to unpack(没有足够的值用于解封)”的 ValueError(值错误)：

```
---------------------------------------

ValueError          Traceback (most recent call last)

<ipython-input-27-080661945d81> in <module>()
----> 1 x,y,z = 2,'hello'
ValueError: not enough values to unpack (expected 3, got 2)
```


4.4.2 索引和切片

可以通过**下标运算符[]**对 tuple 对象执行索引和切片操作。例如：

```
t = ("alpha",2,3.14,[5,'hello'], ("delta", "omega"))
print(t[-1])
print(t[1:4])
print(t[:-1])
```

输出：

```
('delta', 'omega')
(2, 3.14, [5, 'hello'])
('alpha', 2, 3.14, [5, 'hello'])
```

4.4.3 tuple 是不可变的

1．tuple 是不可变的

和 str 字符串一样，不能通过下标修改 tuple 对象中的元素。例如：

```
t = (2,5,8)
t[0] =  5
```

产生“tuple 对象不支持对其数据元素的赋值”的 TypeError(类型错误)：

```
---------------------------------------------------------
TypeError             Traceback (most recent call last)
<ipython-input-6-adf0143216c3> in <module>()
     1 t = (2,5,8)
----> 2 t[0] =  5
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
TypeError: 'tuple' object does not support item assignment
```

可以用 del 删除指向 tuple 对象的变量，减少该对象的引用计数。例如：

```
del t
```

不能修改一个 tuple 对象是指，不能修改其每个元素的值，这些元素的值实际上也是实际对象的引用，如果 tuple 中的某个元素引用了一个可修改的对象，则可以修改它引用的对象，尽管这个元素本身不能修改。例如：

```
a = (2,3.14,'hello',[5,6,'world'])
for e in a:
   print(id(e),end = ' ')
a[3][0]= 13      # a[3]指向的 list 对象[5,6,'world']是可修改的
a[3].append(8)
print()
print(a)
for e in a:
   print(id(e),end = ' ')
```

输出：

```
1767534080 2226147878400 2226148698072 2226130946440
(2, 3.14, 'hello', [13, 6, 'world', 8])
1767534080 2226147878400 2226148698072 2226130946440
```


tuple 的元素是对象的引用，如图 4-2 所示。

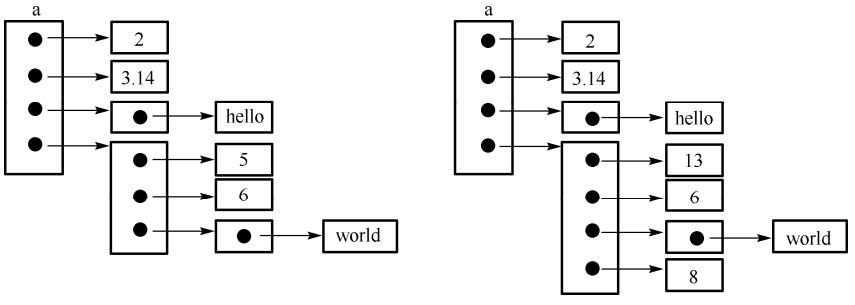


图 4-2　tuple 的元素是对象的引用

可以看到，tuple 对象 a 的每个元素 e 的 id(e)并没有改变，即 a 的每个元素的值并没有改变，也就是 a 本身没有任何改变，修改的是其第 4 个元素引用的 list 对象，该 list 对象从[5,6,'world']修改为[13, 6, 'world', 8]。

如果一个元素指向的对象不可修改，则不能修改这个对象。下列代码试图修改第 3 个元素引用的字符串'hello'：

```
a[2][0]='A'    # a[2]指向的字符串对象'hello'是不可被修改的
```

产生了“str 对象不支持赋值”的 TypeError(类型错误)：

```
---------------------------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-15-d54681727fcb> in <module>()
----> 1 a[2][0]='A'    # a[2]指向的字符串对象'hello'是不可修改的

TypeError: 'str' object does not support item assignment
```

2．+和*运算符

和 str、list 一样，可以用+运算符拼接两个 tuple 对象，而 * 运算符用于重复 tuple 对象。例如：

```
t = (1,2,3) +(5,6,7)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print(t)
t = (1,2,3)*3
print(t)
```

输出：

```
(1, 2, 3, 5, 6, 7)
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

3．成员运算符 in 和 not in

和 str、list 一样，运算符 in 和 not in 分别用于判断一个元素是否“在”或“不在”一个 tuple 对象里。例如：

```
t = (2,3.14,'hello',[5,6])
print(3.14 in t)
print(3 not in t)
```

输出：

```
True
True
```

4．for 遍历 tuple

和 str、list 一样，可以用 for…in 遍历一个 tuple 对象。例如：

```
for name in ('Zhang','Wang'):
    print("Hello,",name)
```

输出：

```
Hello, Zhang
Hello, Wang
```

4.4.4 用内置函数对 tuple 操作

可以用内置函数(max()、min()、sum()等)对一个 tuple 对象进行操作。例如：

```
t= (23,19,2,50,11)
if 19 in t:
   print('19 is in tuple t')
print('max,min and sum of tuple t is:',max(t),min(t),sum(t))

for e in t:
   print(e,end=' ')
print()

print(2*t)
t2= tuple(range(3,10, 2))    #t2 = (3, 5, 7, 9)
print(t2)
print(t+t2)
```

输出：

```
19 is in tuple t
max,min and sum of tuple t is: 50 2 105
23 19 2 50 11
(23,19,2,50,11,23,19,2,50,11)
(3,5,7,9)
(23, 19, 2, 50, 11, 3, 5, 7, 9)
```

4.4.5 tuple 的方法

tuple 同样有一些方法，如 count()方法用于统计某个元素出现了多少次，index()方法用于返回

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

某个元素的下标。例如：

```
t = ('p','y','t','h','o','n','t',)
# count: 某个元素出现了多少次
print(t.count('t'),end = ' ')
# Index: 返回某个元素的下标
print(t.index('y'),end = ' ')
```

输出：

```
2 1
```

总结

- 创建 tuple 字典对象可以用圆括号、内置函数 tuple()。不能用解析式直接创建 tuple 对象，但可以将解析式作为 tuple()的函数参数用于创建 tuple 对象。
- 不同于 list，tuple 对象不可修改。
- 对 list 的操作大部分也可以用于 tuple。例如，下标运算符访问一个或多个元素；成员运算符 in 和 not in；比较运算符；+和*运算符；for 循环；内在函数 max()、min()、sum()；tuple 类的自身方法，如 count()、index()等。

4.5 集合

集合(set)是无序的不重复元素容器类型。可以将一个 set 当作一个 list，但需注意如下几个问题。

- set 中的元素没有顺序，所以无法通过下标访问其中的元素，但可以检查一个值是否在这个集合里。
- set 中不允许出现重复的元素。因此，可以利用 set 对 list 或 tuple 中的元素去重(去除重复元素)。
- 可以对 set 执行数学上的并、交、差、对称差等运算。

如果不注重元素是如何存储(如是否有序)的，就可以考虑用 set 代替 list 来存储数据元素。因为 set 的实现是根据一个元素的值计算出一个哈希值(哈希地址，也称散列值或散列地址)来确定元素的内存存储地址的，从而可以直接根据这个哈希地址访问这个数据元素，要比逐个依次查找快得多，理想的情况是时间复杂度是 O(1)，即一次计算哈希地址就能立即找到这个元素。

4.5.1 创建 set 对象

将所有用逗号,隔开的数据元素放在一对花括号{}里，就创建了一个 set 对象。例如：

```
s = {2, 3.14, 'hello'}
type(s)
```

输出：

```
set
```

集合中的元素必须是“**可哈希的(hashable)**”，list 对象是不可哈希的，因此集合中不能包含 list 对象。例如：

```
s2 = {[2,3.14],'hello'}
```

产生“unhashable type: 'list'(不能哈希的类型:list)”的 TypeError(类型错误)：

```
---------------------------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-4-da76fe391c1f> in <module>()
----> 1 s2 = {[2,3.14],'hello'}
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
TypeError: unhashable type: 'list'
```

和内置函数 list()、tuple()可分别用于创建 list、tuple 对象一样，也可以用内置函数 set()创建 set 对象，其参数可以是一个 list、tuple、字符串、另一个 set 对象或其他可迭代对象。

内置函数 set()作用于一个 list 对象。例如：

```
s1 = set([2,5,8])   #set()接收一个 list 对象
print(s1)
type(s1)
```

输出：

```
{8, 2, 5}
set
```

set 函数作用于一个 tuple 对象。例如：

```
s2 = set((3,6,9))  #set()接收一个 tuple 对象
print(s2)
type(s2)
```

输出：

```
{9, 3, 6}
set
```


set 函数作用于一个 set 对象或字符串对象。例如：

```
s3 = set({1,4,7})    #set()接收一个 set 对象
s4 = set('python')   #set()接收一个 str 对象
print(s3)
print(s4)
```

输出：

```
{1, 4, 7}
{'o', 'p', 'n', 'h', 'y', 't'}
```

同样，也可以给函数 set()传递一个 list 解析式。例如：

```
s5 = set([x*2+1 for x in range(3, 9)])  #set()接收一个 list 解析式生成的 list 对象
print(s5)
```

输出：

```
{7, 9, 11, 13, 15, 17}
```

如果创建 set 时传递的值重复，则 set 会自动去除重复的元素。例如：

```
s6 = {3,5,7,11,3,5,9}
print(s6)

s7  = set([7,11,2,7,8])
print(s7)
```

输出：

```
{3, 5, 7, 9, 11}
{8, 2, 11, 7}
```

创建一个空的 set 对象时，不能像下面这样：

```
es = {}
type(es)
```

输出：

```
dict
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

这实际上创建的是一个 dict(字典)对象。如果要创建一个空的 set 对象，则需要用内置函数 set()。例如：

```
es = set()
type(es)
```

输出：

```
set
```

4.5.2 遍历 set

像 list、tuple、str 等容器一样，可以用 for 循环遍历 set 对象的所有元素。例如：

```
for e in s:
   print(e, end= " ")
2 3.14 hello
```

4.5.3 用内置函数对 set 操作

和 list、tuple 一样，可以用内置函数，如 len()、max()、min()、sum()等对 set 对象进行操作。例如：

```
print(s7)
print(len(s7))
print(min(s7))
print(max(s7))
print(sum(s7))
print(id(s7))
```

输出：

```
{8, 2, 11, 7}
4
2
11
28
2859915094504
```

4.5.4 set 的方法

可以用 set 类的方法对 set 对象进行处理。例如：

```
s8 = {'python','hello','world','小白'}
print(s8)
print(id(s8))

s8.add(3.14)          #用 add()方法添加一个对象
print(s8)
print(id(s8))

s8.remove('hello')    #用 remove()方法删除一个对象
print(s8)
```

输出：

```
{'python', 'hello', '小白', 'world'}
2859932459752
{3.14, '小白', 'hello', 'world', 'python'}
2859932459752
{3.14, '小白', 'world', 'python'}
```

其中，用 add()方法添加对象，用 remove()方法删除一个对象。如果试图用 remove()方法删除一个不存在的值：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
s8.remove(2)
```

则会产生 KeyError 异常：

```
------------------------------------------------------------

KeyError                  Traceback (most recent call last)

<ipython-input-24-2c876338ee3a> in <module>()
----> 1 s8.remove(2)
KeyError: 2
```

如果不希望 remove()方法抛出异常，则可以用 discard()方法代替 remove()方法。用 discard()方法删除一个不存在的值时，不会抛出异常。例如：

```
s8.discard(2)
```

可以用 update()方法一次添加多个值。update()方法的参数是一个可迭代对象，如 list、tuple、str 等类型对象。例如：

```
s9 = {1,2,3}
s9.update('hello')
print(s9)
s9.update([23,17])
print(s9)
```


输出：

```
{1, 2, 3, 'o', 'e', 'l', 'h'}
{1, 2, 3, 'o', 'e', 17, 'l', 'h', 23}
```

如果要删除一个 set 对象的所有元素，则可以用 clear()方法。例如：

```
s9.clear()
print(s9)
```

输出：

```
set()
```

issubset()方法和 issuperset()方法可以用来判断一个 set 对象表示的集合是否是另一个 set 对象表示的集合的子集或超集。

如果 A 是 B 的子集，那么 B 就是 A 的超集，反之亦然。例如：

```
A = {1,2,3}
B = {1,2,3,4,5}
print(A.issubset(B))
print(B.issuperset(A))
print(B.issubset(A))
```

输出：

```
True
True
False
```

isdisjoint()方法用于判断两个集合是否有交集，即有两个集合 A、B，只有当 A&B 是非空集合时，A.isdisjoint(B)的值才是 True。例如：

```
print(A.isdisjoint(B))
C = {4,5}
print(A.isdisjoint(C))
```

输出：

```
False
True
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

4.5.5 set 的运算符操作

可以用运算符来比较两个 set 对象表示的集合是否具有子集或超集的关系。例如：

```
print(A<B)        # A 是 B 的子集吗?
print(B>A)        # B 是 A 的超集吗?
print(A>B)
```

输出：

```
True
True
False
```

==和!=可以分别用来判断两个 set 对象集合是否相等。另外，还有>=或<=等运算符也可以达到同样效果。例如：

```
C ={1,2,3}
print(A==B)
print(A==C)
print(A!=B)
print(A!=C)
print(A<=B)
print(A>=B)
```

输出：

```
False
True
True
False
True
False
```

in 和 not in 可以分别用来判断一个值是否在一个 set 对象集合里。例如：

```
if 3 in A:
   print('3 in set A')

if 6 not in A:
   print('6 not in set A')
```

输出：

```
3 in set A
6 not in set A
```

4.5.6 set 的集合运算（并、交、对称差）

(1) 并运算：可以通过 set 的 union() 方法，也可以通过**或运算符 |** 进行并运算。例如：

```
A= {1,2,3}
B = {4,5,6}
C = A.union(B)  # 等价于
print(A)
print(C)
D = A|B
print(D)
```

输出：

```
{1, 2, 3}
{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 6}
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

(2)交运算：可以通过 set 的 intersection()方法，也可以通过**与运算符&**进行交运算。例如：

```
A= {1,2,3,4}
B = {4,5,6}
C = A.intersection(B)
D = A&B
print(C)
print(D)
```

输出：

```
{4}
{4}
```

(3)差运算：可以通过 set 的 difference()方法，也可以通过**减法运算符–**进行差运算。例如：

```
A= {1,2,3,4}
B = {4,5,6}
C = A.difference(B)
D = A-B
print(C)
print(D)
```

输出：

```
{1, 2, 3}
{1, 2, 3}
```


(4)对称差运算：两个集合的**对称差**是只在两个集合中的一个集合出现的值构成的集合。例如：A= {1,2,3,4} , B = {3,4,5,6}，则 A 和 B 的对称差为{1,2,5,6}。

可以通过 set 的 symmetric_difference()方法，也可以通过对称差运算符^进行对称差运算。例如：

```
A= {1,2,3,4}
B = {3,4,5,6}
C = A.symmetric_difference(B)
D  =A^B
print(C)
print(D)
```

输出：

```
{1, 2, 5, 6}
{1, 2, 5, 6}
```

总结

- set(集合)是无序的不重复元素的容器类型。集合中的元素没有顺序之分，下标运算符不能作用于 set 对象。
- 作为容器对象，可用 in 和 not in 分别检查元素是否在集合里。可用 for 循环遍历其中的元素。
- 对 set 对象可执行数学上的并、交、差、对称差等运算。可用比较运算符(<、>、==等)比较两个 set 对象的关系。
- 可用内置函数或 set 的方法对 set 对象进行操作。例如，函数 sum()求和，add()和 remove()方法分别可添加、删除元素等。update()可以添加 1 个或多个对象。

4.6 字典

字典(dict)是一种通过键存储或查询值的数据类型，是一个“键-值(key-value)”映射的可变容器模型。每个键-值对的键和值之间用冒号 : 隔开，不同的键-值对之间用逗号 , 隔开。整个字典的

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

所有键-值对包括在花括号{}中。语法格式如下：

```
{key1 : value1, key2 : value2,... }
```

一个 dict 对象包含多个键-值对，这些键-值对的键(key)是唯一的，并且是可以用哈希算法计算出哈希地址的。因此，通过键(key)算出其存储地址，就可以立即查询到相应的值，因此，和 set 一样，dict 具有非常快的查找速度。

4.6.1 创建字典对象

可以用花括号{}、内置函数 dict()、字典解析式创建一个 dict 对象。例如：

```
#创建空的 dict 对象
d= {}
d1 = dict()

#用{}创建有初始值的 dict 对象
d1 = {'李平':78.5,'张伟':80,'赵四':90}
d2 = {'李平': 78.5, 1: [2, 4, 3]}           #key 既可以是字符串也可以是整数

#用函数 dict()创建有初始值的 dict 对象
d3 = dict({'李平': 78.5, 1: [2, 4, 3]})     #直接传入一个 dict 对象
d4 = dict(LiPing =78.5, 张伟= [2, 4, 3])    #按照关键字参数传递键值

#用"字典解析式"创建 dict 对象
d5 = {x: x**3 for x in range(5)}
print(d)
print(d1)
print(d2)
print(d3)
print(d4)
print(d5)
```

输出：

```
{}
{'李平': 78.5, '张伟': 80, '赵四': 90}
{'李平': 78.5, 1: [2, 4, 3]}
{'李平': 78.5, 1: [2, 4, 3]}
{'LiPing': 78.5, '张伟': [2, 4, 3]}
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}
```

注意

用 dict()构造 dict 对象时，如果按照关键字参数传递键值，则其关键字必须是一个有效的标识符而不能是其他表达式。例如：

```
d6 = dict('LiPing'='John')    #'LiPing'是一个表达式
d7 = dict(1= [2, 4, 3])       #1 是一个表达式
```

产生下列错误：

```
  File "<ipython-input-41-35462deb4cc5>", line 1
    d6 = dict('LiPing'='John')    # 'LiPing'是一个表达式
              ^
SyntaxError: keyword can't be an expression
```

另外，**fromkeys**()方法也可以用于创建 dict 对象，其格式如下：

```
dict.fromkeys(iterable, value=None)
```

其中，iterable 是键的可迭代对象，而 value 是可选参数，表示键对应的值，默认值为 None。这

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

种方式创建的 dict 对象所有键对应的值都是一样的。例如:

```
keys = ('python', 1)
d = dict.fromkeys(keys)            #创建两个键分别为'python'和 1 的 dict 对象
print(d)
d2 = dict.fromkeys(keys, 'hello') #每个键对应的值默认都是'hello'
print(d2)
```

输出:

```
{'python': None, 1: None}
{'python': 'hello', 1: 'hello'}
```

4.6.2 获取键的值

获取键的值的方法是用**下标运算符**[]，将键传入，以得到该键对应的值。例如：

```
d = {'李平': 78.5, 1: [2, 4, 3]}
print(d['李平'])
print(d[1])
78.5
[2, 4, 3]
```

但如果不存在这个键，则下标运算符[]就会抛出 KeyError 错误。例如：

```
print(d[2])
产生 KeyError 错误:
---------------------------------------------------------------------------

KeyError                Traceback (most recent call last)

<ipython-input-44-c8f93a31d4a2> in <module>()
----> 1 print(d[2])
KeyError: 2
```

为了避免抛出错误，还可以用 get()方法代替下标运算符[]得到一个键对应的值，如果没有这个键，就返回该函数第 2 个参数设置的默认值。例如：

```
D.get(k[,d])
```

如果键 k 在 dict 对象 D 中，则返回 D[k]的值；否则，返回可选参数 d 的值，默认为 None。例如：

```
d = {'李平': 78.5, 1: [2, 4, 3]}
print(d.get('李平'))
print(d.get(2,'默认值'))
```

输出：

```
78.5
默认值
```

4.6.3 通过下标插入或更新一个键值

可以通过下标插入或更新一个键值。例如：

```
d = {'李平': 78.5, 1: [2, 4, 3]}
d['李平']= ('男',23)        #更新一个存在键值
print(d)
d['赵薇']= ('女',32)        #键不存在，则插入新的键-值
print(d)
```

输出：

```
{'李平': ('男', 23), 1: [2, 4, 3]}
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
{'李平': ('男', 23), 1: [2, 4, 3], '赵薇': ('女', 32)}
```

4.6.4 插入或更新多个键值：update()方法

update()方法也可以插入或更新一个或多个键-值。例如：

```
d = {'李平': 78.5}
d.update({'李平': ('男', 23)})
print(d)
d.update(dict({'李平': 90}))        #传递一个 dict 对象
print(d)
d.update(李平= ('学生', 21))        #传递关键字参数
print(d)

d.update({'python': 3.14})          #如果键不存在，则插入新的键-值
print(d)
d.update({'python': 60,'C':90})     #可以传递多个键-值用于更新或插入
print(d)
```

输出：

```
{'李平': ('男', 23)}
{'李平': 90}
{'李平': ('学生', 21)}
{'李平': ('学生', 21), 'python': 3.14}
{'李平': ('学生', 21), 'python': 60, 'C': 90}
```

4.6.5 删除键值

可以用 del()或 pop()方法删除一个键对应的键-值对，其中 pop()方法会返回被删除的键对应的值。例如：

```
d = {'李平': 78.5, 1: [2, 4, 3], 'python': 60}
del d['李平']     #删除 key 为'李平'的键-值对
print(d)
d.pop(1)          #删除 key 为 1 的键-值对
print(d)
```

输出：

```
{1: [2, 4, 3], 'python': 60}
{'python': 60}
```

如果删除不存在的键，则会抛出错误异常。例如：

```
del d[2]
```

产生 KeyError(键错误)：

```
---------------------------------------------------------------------------

KeyError     Traceback (most recent call last)

<ipython-input-62-31e076e1b505> in <module>()
----> 1 del d[2]
KeyError: 2
```

可以用 clear()方法删除一个 dict 中所有的键-值对。例如：

```
d = {'李平':78.5,'张伟':80}
d.clear()
print(d)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
{}
```

4.6.6 获取所有键、所有值、所有键值

集合 dict 的 keys()、values()、items()方法可分别返回一个 dict 对象的所有键、所有值、所有键值对构成的可迭代对象。

```
d = {'a':1,'b':2,'c':3}
print(d.keys())
print(d.values())
print(d.items())
```

输出：

```
dict_keys(['a', 'b', 'c'])
dict_values([1, 2, 3])
dict_items([('a', 1), ('b', 2), ('c', 3)])
```

4.6.7 遍历所有键、所有值、所有键值

用 for…in 循环可遍历一个 dict 对象的每个键，或遍历 keys()、values()或 items()方法返回的包含键、值或键值的可迭代对象。

```
print('遍历每个键:')
for key in d:
   print(key,end = ',')
print()print('遍历每个键:')
for key in d.keys():
   print(key,end = ',')
print()

print('遍历每个值:')
for value in d.values():
   print(value,end = ',')
print()

print('遍历每个键值对：')
for key,value in d.items():
   print(key,value,sep ='-',end = '\t' )
```

输出：

```
遍历每个键:
a,b,c,
遍历每个键:
a,b,c,
遍历每个值:
1,2,3,
遍历每个键值对:
a-1 b-2 c-3
```

4.6.8 用内置函数访问 dict 对象

内置函数 len()可查询 dict 对象的元素个数，内置函数 str()可将一个 dict 对象转化为一个字符串表示。

```
d = {'李平': 78.5, 1: [2, 4, 3], 'python': 3.14}
print(len(d))     #用 len()查询 dict 的数据元素(键值对)的个数
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print(str(d))      #str()得到一个 dict 对象的字符串表示
```

输出：

```
3
{'李平': 78.5, 1: [2, 4, 3], 'python': 3.14}
```

4.6.9 从两个可迭代对象创建一个 dict

函数 dict()还可以接收两个可迭代对象，以创建一个新 dict 对象。其格式如下：

```
new_dict = dict(zip(keys, values))
```

其中，keys、values 是两个可迭代对象(如 list、tuple、str 对象)，dict()返回创建的 dict 对象。例如：

```
d = dict(zip(('a','b','c'),[1,2,3]))
print(d)
```

输出：

```
{'a': 1, 'b': 2, 'c': 3}
```

4.6.10 用 in 检测 dict 对象是否包含某个键

可以用 in 检测 dict 对象是否包含某个键，但无法检测某个值是否在 dict 对象中。

```
d = {'a':1,'b':2,'c':3}
print('a' in d)
print(3 not in d)
```

输出：

```
True
True
```

总结

- 创建 dict 对象有四种方法：花括号{}、内置函数 dict()、fromkeys()方法、字典解析式。
- 对于一个 dict 对象，下标运算符[]或 update()方法可以通过键(key)查询、修改、删除、添加元素。
- 用方法 keys()、values()、items()可以分别获得一个 dict 对象的所有键、所有值、所有元素。可以迭代访问其中的键(keys)、值(values)或键值对(key-values)。
- 可以通过内置函数或 dict 的方法操作一个 dict 对象。
- 可以用 in 检测某个键是否在 dict 对象里，但不能检测某个值是否在 dict 对象中。

4.7 用强化学习 Q-Learning 算法求解最佳路径

4.7.1 强化学习

机器学习方法主要分为监督式学习、非监督式学习和强化学习。在监督式学习中，每个样本都有一个正确的答案，通过许多这种答案已知的样本(x_i,y_i)学习一个函数$y=(x)$，其中的 x 是样本特征，而 y 就是正确的答案，如前面讲述的线性回归预测房屋价格就是一个典型的监督式学习案例。监督式学习已经在许多领域，如人脸识别、目标检测、分类、机器翻译、下棋等取得了成功。监督式学习是在和世界冠军对抗中取得全胜的围棋程序 AlphaGo(阿尔法狗)的核心技术。然而，现实中的许多序列决策和控制问题很难用监督式学习去解决。例如，要开发一个智能机器人去探索某个环境(如迷宫)，初始时，没有一个正确的行动答案告诉机器人如何行走，即对于一个位置(状态)，到底向哪

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

个方向(东南西北)走没有一个明确的答案。再如，在 Pong 游戏中，挡板可以有向上、向下和不动三个动作，但很难知道某一时刻到底应该采用哪个动作可导致最终的成功。最终的成功或失败是由许多一系列的动作完成的，很难知道其中每个动作是否正确。这类决策和控制问题就需用到强化学习方法。强化学习方法是通过不断地迭代和试错来学习决策的。

在强化学习中，虽然每个决策动作没有明确的正确与否的标签，但通常会得到来自环境的“奖励”。例如，小孩学习走路，就是一个不断的试错学习的过程，如果失败了就会摔倒，如果成功了就会继续向前。通过这些正或负的“奖励”，小孩会逐渐调整走路的动作。

另外，监督式学习(如深度学习)需要准备大量的答案已知的样本，这通常是一个非常耗时、耗力的工程，这是一个封闭的、与外部环境隔绝的学习过程，而强化学习通过与环境的交互，根据环境的反馈和不断试错，是适应动态的、不确定因素的学习，不仅可以解决监督式学习不能解决的问题，而且也不需要大量样本，更符合人类的学习过程。实际上，Google 公司的新一代的阿尔法元(AlphaZero)已经完全采用了强化学习，它不需要人类棋手的棋盘知识就能完胜之前已完胜人类围棋冠军的阿尔法狗。Google DeepMind 和 Facebook AI Research 等著名研究机构已经将强化学习用于许多领域，如玩各种游戏、股票预测等。

如图 4-3 所示，是一个机器人从任意状态出发寻找金币的游戏。这是一个典型的强化学习问题，机器人在任意时刻都位于某个位置或某个状态，它可以采取一些动作(向东、向南、向西、向北)，找到金币则获得奖励 1，碰到海盗则损失 1，如果没有遇到金币或强盗则奖励为 0；找到金币或者碰到海盗，则机器人停止。

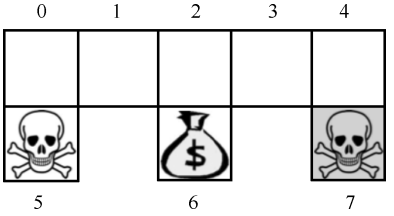


图 4-3　机器人寻金币

强化学习中的状态、决策、状态转移、奖励等可以用**马尔可夫决策过程**(**Markov Decision Processes，MDP**)来刻画。MDP 可以用一个五元组(S、A、P_{sa}、γ、R)表示。

- S 是所有状态的集合。例如，机器人寻宝中机器人的位置。
- A 是所有动作的集合。例如，机器人的(向东、向南、向西、向北)的动作。
- $P_{sa}(s')$ 是状态转移概率，给出了从状态 s 和该状态下的动作 a 转移到下一个状态 s' 的概率。
- $\gamma \in [0,1]$ 是折扣因子，表示未来奖励对于当前动作的作用大小。
- R 是一个 $S \times A \to \mathbb{R}$ 的奖励函数，即在状态 s 下采取动作 a 所得到的直接奖励。

MDP 决策是一个动态过程：从一个初始状态 s_0 出发，采取一个动作 a_0，可能过渡到下一个状态 s_1，在 s_1 状态下采取一个动作 a_1，可能过渡到下一个状态 s_2、…，这个过程一直进行，直到遇到最终状态(成功或失败)。如下所示：

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

在访问状态序列 s_0、s_1、s_2、… 的过程中采取的对应动作序列 a_0、a_1、a_2、… 会得到环境的反馈“奖励”，这些奖励累加起来就构造了从初始状态 s_0 出发的总奖励：

$$R(s_0,a_0)+\gamma R(s_1,a_1)+\gamma^2 R(s_2,a_2)+\dots$$

其中的折扣因子 γ 表示长期奖励的重要性，如果该值比较小，则说明更关注短期的奖励；如果该值比较大，则说明长期奖励也很重要，也就是用于平衡眼前利益和长远利益。如果 γ 比较小，则说明今天奖励 1 元钱比将来奖励 1 元钱更重要。

因为从一个初始状态出发可能采取很多不同的动作序列，所以强化学习的目标是最大化如下的平均总奖励：

$$E_{s_0,a_0,s_1,a_1,\dots}[R(s_0,a_0)+\gamma R(s_1,a_1)+\gamma^2 R(s_2,a_2)+\dots]$$

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

即各种可能的状态动作序列的奖励的期望(平均值)。

policy(策略)是一个状态到动作的函数： $\pi : S \to A$ 。策略规定了在某个状态 s 应采取哪个动作 a，即 $a = \pi(s)$ 。状态的价值函数 $V^{\pi}(s)$ 是在策略 π 下从状态 s 出发的平均总奖励：

$$V^{\pi}(s) = E_{s_0,a_0,s_1,a_1,\ldots}[R(s_0,a_0) + \gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \ldots \mid s_0 = s, \pi]$$

即从初始状态 s_0 出发的各种可能的状态动作序列的奖励的期望(平均值)。

对于一个固定的策略 π ，价值函数满足**贝尔曼方程**(Bellman Equations)：

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P_{s,a}(s') V^{\pi}(s')$$

其中的 $R(s)$ 是 s 状态下采用各种动作的期望直接奖励。

可以定义一个状态的最优价值函数，即任意状态 s 的最优价值：

$$V^{*}(s) = \max_{\pi} V^{\pi}(s)$$

可以证明，这个最优价值函数也满足贝尔曼方程：

$$V^{*}(s) = R(s) + \max_{a} \gamma \sum_{s' \in S} P_{s,a}(s') V^{*}(s')$$

定义如下的策略 $\pi^{*} : S \to A$ ：

$$\pi^{*}(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{s,a}(s') V^{*}(s')$$

即在一个状态 s 下，选择一个动作 a，使得在所有动作中执行该动作后的平均价值最大。对任意状态 s，按照这个 π^{*} 策略选择的动作 $a = \pi^{*}(s)$ 就能使状态 s 的价值取得最优值。也就是这个策略就是最优的策略。所以只要能求得最优价值函数，在任意状态 s，根据这个公式，从一个状态可能动作 $a \in A$ 中选择一个使 $\sum_{s' \in S} P_{s,a}(s') V^{*}(s')$ 值最大的动作 a，就是一个最优策略。

上述公式涉及概率，看起来复杂，初学者一时不能很好地理解，但并不影响其对 Q-Learning 算法的学习和实现。

4.7.2 Q-Learning 算法

1. Q-Learning 算法

Q-Learning 算法是一种求 MDP 问题的最佳策略的方法。它不是计算一个状态的价值，而是通过计算(状态、动作)的价值来寻找最佳的策略，即用 $Q(s,a)$ 描述状态 s 下执行动作 a 的价值，也称质量。对于最佳的策略 Q^{*}，同样满足贝尔曼方程：

$$Q^{*}(s,a) = R(s) + \gamma \max_{a'} Q^{*}(s',a' \mid s,a)$$

即 $Q^{*}(s,a)$ 等于平均直接奖励和其后续状态 s' 下的各个动作 a' 中的最大 $Q^{*}(s',a')$ 值的一个折扣值，也就是 $\gamma \max_{a'} Q^{*}(s',a')$ 之和。

和求解方程的根的迭代法一样，为了求解这个最佳的 $Q^{*}(s,a)$，Q-Learning 算法通过一个简单的“时差更新”方法来迭代地逼近最佳的 $Q^{*}(s,a)$ 。

首先初始化所有的 $Q^{*}(s,a)$ 值为一个随机初始值(如 0)，然后用下面的迭代公式不断更新 $Q(s,a)$ 的值：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

$$Q^{\text{new}}(s,a)=Q(s,a)+\alpha[r+\gamma\max_{a'}Q(s',a')-Q(s,a)]$$

其中的 r 是在状态 s 采取动作 a 的直接奖励，$r+\gamma\max_{a'}Q(s',a')$ 就是根据后续(状态、动作)价值 $Q(s',a')$ 计算出来的 $Q(s,a)$ 的预测价值，用这个预测价值和原来 $Q(s,a)$ 的误差 $r+\gamma\max_{a'}Q(s',a')-Q(s,a)$ 来更新修改 $Q(s,a)$，使得 $Q(s,a)$ 尽量趋向这个更大的价值 $r+\gamma\max_{a'}Q(s',a')$。$a$ 是学习率，表示学习的速度，其值大则表示更快地向值 $r+\gamma\max_{a'}Q(s',a')$ 靠拢；其值小则表示慢一点靠拢。可以将原来的 $Q(s,a)$ 和 $r+\gamma\max_{a'}Q(s',a')$ 看成实数轴上的两个点，γ 控制 $Q^{\text{new}}(s,a)$ 靠近哪个更多些。

Q-Learning 算法通常任意选取一个状态 s_0 出发，然后采用 ϵ 贪婪法选择一个动作 a_0，得到一个奖励 r_0 并过渡到一个新的状态 s_1，然后根据上述迭代公式更新 $Q(s_0,a_0)$，再从 s_1 出发，采用 ϵ 贪婪法选择一个动作 a_1，得到一个奖励 r_1 并过渡到一个新的状态 s_2，然后根据上述迭代公式更新 $Q(s_1,a_1)$，…，这个过程一直进行直到遇到最终状态，构成一个完整的探索序列：

$$\langle s_0,a_0,r_1,s_1,a_1,r_2,s_2,\ldots\rangle$$

这个探索序列也称 **episode**(片段)。通常需要经过很多次 episode，$Q(s,a)$ 才能收敛到最佳的 $Q^*(s,a)$。

Q-Learning 算法的过程如下：

```
初始化 Q(s,a)=0
多次(如 200 次)episode:
   对每个 episode 选择一个出发状态 s，执行下面的循环:
      用 ε 贪婪法选择一个动作 a'
      得到环境反馈的(r,s')
         如果 s'不是结束状态，则更新 Q(s,a)，s = s'；否则，这次 episode 结束
```


2．ε 贪婪法

在一个状态 s 下，贪婪法总是在可选的动作 a 中选择一个 $Q(s,a)$ 最大的动作，以便最大化最终的总奖励。然而，因为开始时，任意(*s*,*a*)的 $Q(s,a)$ 并没有达到最大值，如果只采用贪婪法，则容易被短期利益所迷惑而陷入局部最大，而不能找到全局最佳的策略。ε 贪婪法的思想是对贪婪法做一点修正，大概率情况下选取当前 $Q(s,a)$ 最大值的动作 a，但也会以较小的概率去随机选择一个动作，从而探索未知的路线。例如，设置 $\varepsilon=0.1$，表示以 0.1 的概率从所有可能动作中任意选择一个可行动作，以 0.9 的概率采取最佳动作。ε 贪婪法可以在利用已有知识和探索未知领域之间取得一个平衡。这个 ε 和学习率 α 一样，需要根据实际问题选择一个合适的值，而且经常会采用自适应的方法，在迭代过程中不断调整。为简单起见，下面的程序采用固定的 ε 和 α 值。

4.7.3 Q-Learning 算法的 Python 实现

Q- Learning 算法实现需要以下两个数据结构：

- 一个 *Q* 表，记录任意状态 s 下的所有动作 a 的 $Q(s,a)$；
- 一个数据结构，是描述环境的模型，即在某状态 s 执行某动作 a 后得到的反馈信息(r,s',is_timinal)，即直接奖励 r、过渡到的新状态 s'、是否到达了终止状态。

Q 表对每个状态 s 都用一个线性表来存储从该状态出发的每个动作 a 所对应的 $Q(s,a)$ 值，如下所示：

$$s\text{: }(Q(s,a_1),\ Q(s,a_2),\ Q(s,a_3),\ \ldots)$$

即用一个状态的所有动作的 $Q(s,a)$ 构成了一个线性表，可称为该状态的 *Q* 表。例如，寻金币问题的第 1 个状态 s=0 对应的 *Q* 表如下：

s=0：(<'*s*',0>，<'*e*',0>)

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

其中的每个元素<*a*, *Q*(*s*,*a*)>都由动作名 *a* 和 *Q*(*s*,*a*)价值构成，如元素<'*s*',0>表示动作名是'*s*'(向南走)，而该动作的价值初始化为 0；元素<'*e*',0>表示动作名是'*e*'(向南走)，而该动作的价值初始化为 0。

所有状态的 *Q* 表存储在另一个线性表中，即整个问题的 *Q* 表，如下：

((<'*s*',0>，<'*e*',0>)，(<'*w*',0>，<'*e*',0>)，(<'*w*',0>，<'*s*',0>，<'*e*',0>)，(<'*w*',0>，<'*e*',0>)，(<'*w*',0>，<'*s*',0>))

对于寻宝问题的 5 个可行状态(非最终状态 *s*)，可以用一个 list 对象存储所有状态的 *Q* 表，而每个状态的 *Q* 表用一个 dict 对象来存储，以“键–值”对即“动作名–价值”的形式表示每个 *Q*(*s*,*a*)。

该问题的 *Q* 表如下：

Q = [{'*s*':0,'*e*':0},{'*w*':0,'*e*':0},{'*w*':0,'*s*':0,'*e*':0},{'*w*':0,'*e*':0},{'*w*':0,'*s*':0}]

每个状态对应 list 的一个 dict 对象表示了这个状态的 *Q* 表。

例如，下标为 1 的状态对应的 *Q* 表是{'*w*':0,'*e*':0}，其中的元素<'*w*':0>表示动作名是'*w*'(向西走)，而该动作的价值初始化为 0，元素<'*e*':0>表示动作名是'*e*'(向南走)，而该动作的价值初始化为 0。

类似地，可以用一个状态转移表描述在状态 *s* 执行动作 *a* 后得到的直接反馈 *r* 和转移到的新状态 *s*'：

T：(*T*(s_1,a_1)，*T*(s_2,a_2，*T*(s_3,a_3)，…)

即记录了所有的(*s*,*a*)执行的直接奖励和状态转移情况。其中的每个 *T*(*s*,*a*)可以描述为：

T(*s*,*a*)：<*s*,*a*,*r*,*s*'>

寻宝问题的状态转移表如下：

T = {(0,'*s*'):(5,–1),(0,'*e*'):(1,0),(1,'*w*'):(0,0),(1,'*e*'):(2,0),(2,'*s*'):(6,1),(2,'*w*'):(1,0),
(2,'*e*'):(3,0),(3,'*w*'):(2,0),(3,'*e*'):(4,0),(4,'*w*'):(3,0),(4,'*s*'):(7,–1)}

每个元素描述了执行(*s*,*a*)产生的反馈(*r*,*s*')，即(*s*,*a*,*r*,*s*')。如果用一个 dict 元素(0,'*s*'):(5,–1)表示在状态 0 执行动作'*s*'，则会过渡到状态 5 并且有一个直接奖励–1。

下面的程序是针对寻宝问题的 Q-learning 算法的具体实现。其中：

辅助函数 buildQtable()用于初始化这个 *Q* 表；

辅助函数 buildTtable()用于初始化并返回状态转移表终止状态集合；

辅助函数 getenvfeedback()根据这个环境数据结构，返回执行当前的(状态、动作)后的奖励和下一个状态，以及是否到达最终状态；

辅助函数 choose_action()采用 ϵ 贪婪法选择一个动作；

辅助函数 randomstartstate()在开始一个 episode 时选择一个出发状态。

函数 Q_Learning()是 Q-learning 算法的实现。函数 Q_Learning()中一共执行了最多 MAX_EPISODES 次探索片段，在每个探索片段中，首先用函数 random_start_state()随机选择一个出发状态，只要未到达最终状态，就用函数 choose_action()选择一个动作，然后通过函数 get_env_feedback()得到环境的反馈(直接奖励、下一个状态、下一个状态是否为终止状态)，并根据下一个状态是否为最终状态，得到当前状态和动作的预测 *Q* 值 q_target，用该值更新当前的 *Q* 值。如果下一个状态不是最终状态，则作为新的当前状态，继续上述过程，直到到达最终状态，结束这次探索片段。

```
# q_Table
def build_Q_table(state_actions = None):
    Q = {}
    if state_actions == None:
        Q =  [ {'s':0,'e':0},{'w':0,'e':0},{'w':0,'s':0,'e':0},
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                    {'w':0,'e':0},{'w':0,'s':0}]
    else:
        for actions in state_actions:
            for action in actions:
                action.append(0)
    return Q


#初始化游戏环境：状态转移及奖励、终止状态
def build_T_table(transit_table = None,terminal_states = None):
    if transit_table == None:
        transit_table =  {(0,'s'):(5,-1),(0,'e'):(1,0),(1,'w'):(0,0),
             (1,'e'):(2,0),(2,'s'):(6,1),(2,'w'):(1,0),(2,'e'):(3,0),
             (3,'w'):(2,0),(3,'e'):(4,0),(4,'w'):(3,0),(4,'s'):(7,-1)}
    if terminal_states==None:
        terminal_states = {5,6,7}
    return transit_table,terminal_states

def get_env_feedback(state, action,transit_table,terminal_states):
    next_state,reward = transit_table[(state,action)]
    is_terminal = next_state in terminal_states
    return next_state,reward,is_terminal

def choose_action(state, Q,EPSILON=0.1):
    action_values = Q[state]
    if random.random() < EPSILON :
        action_name = random.choice(list(action_values))
    else:
        max_elem = max(action_values,key=action_values.get)
        action_name  = max_elem[0]
    return action_name

import random
def random_start_state(size,terminal_states):
    while True:
        s = random.randint(0,size-1)
        if s not in terminal_states:
            return s

def Q_Learning(Q,transit_table ,terminal_states,
             MAX_EPISODES = 15,EPSILON = 0.2,ALPHA = 0.1,GAMMA = 0.9):

    for episode in range(MAX_EPISODES):  #循环的次数
        step_counter = 0
        s = random_start_state(len(Q),terminal_states)
        s  = 4
        is_terminated = False
        while not is_terminated:        #循环直到一局游戏结束
            action = choose_action(s, Q,EPSILON)  # 根据状态选择动作
# 获取环境的反馈
            s_next, R, is_terminated = get_env_feedback(s, action,
                   transit_table,terminal_states)
            q_predict = Q[s][action]
            if not is_terminated:       #如果没有结束则更新 q_target 值
                action_values = Q[s_next]
                #max_action = max(action_values,key=action_values.get)
                q_target = R + GAMMA * action_values[max(action_values,
                        key=action_values.get)]
            else:                       #S_是结束状态
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                q_target = R

            Q[s][action] += ALPHA * (q_target - q_predict)  #更新Q值
            s = s_next  # 进入下一个状态
        return Q

    Q = build_Q_table()
    transit_table,terminal_states = build_T_table()
    Q = Q_Learning(Q,transit_table,terminal_states,100)
    print('\r\nQ-table:\n')
    print(Q)
```

通过 Q-Learning 算法求得 Q 表后，就知道了任意状态 s 下执行动作 a 的价值 $Q(s,a)$。如果要求从任意状态 s_0 出发到达目的地的最佳路径，就可以从 s_0 的所有可能的动作 a 中选择 $Q(s_0,a)$ 最大的那个动作 a_0 执行，然后到达下一个状态 s_1，在此状态的所有可能的动作 a 中再选择并执行一个使 $Q(s_1,a)$ 最大的动作 a_1，这样可以一直到达目标位置，从而得到一个最佳路径：

```
    def shortest_path(state,Q,transit_table,terminals):
        path = []
        count=0
        while state not in terminals:
            path.append(state)
            action_values = Q[state]
            action = max(action_values,key=action_values.get)
            s_next,reward = transit_table[(state,action)]
            state = s_next
        path.append(state)
        return path

    s = 0
    path = shortest_path(s,Q,transit_table,terminal_states)
    print("path:\n",path)
```

执行程序，输出：

```
    path: [0, 1, 2, 6]
```

可以看到，从初始状态 s=0 出发，需要经过状态 1、2 到达目标状态 6。

这个 Q-Learning 算法实现不同于网上针对特定问题的特定实现，除两个辅助函数 buildQtable() 和 buildTtable() 是针对寻宝问题的，其他的函数都可用于其他的类似问题。例如，网上的“走出 Q-Learning”的房间问题、迷宫问题、QL 玩 FlappyBird 游戏等。因此，这是一个通用的 Q-Learning 算法程序实现。因篇幅有限，本书仅以迷宫问题为例，说明程序的通用性。

针对迷宫问题，只需要初始化 Q 表和状态转移表（包括终止状态集合）就可以了。这里用一个辅助函数 init_game_maze() 从一个迷宫的二维数组初始化程序的 Q 表和状态转移表（包括终止状态集合）：

```
    def init_game_maze(maze):
        m = len(maze)
        n = len(maze[0])
        s = 0
        Q = []                  #Q表
        T = dict()              #状态转移表
        terminals = set()
        for i in range(m):
            for j in range(n):
                Q.append(dict())
                if i>=1:
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                    s_ = s-n
                    Q[s]['U'] = 0
                    T[(s,'U')] = (s_,maze[i-1][j])
                if i<m-1:
                    s_ = s+n
                    Q[s]['D'] = 0
                    T[(s,'D')] = (s_,maze[i+1][j])
                if j>=1:
                    s_ = s-1
                    Q[s]['L'] = 0
                    T[(s,'L')] = (s_,maze[i][j-1])
                if j<m-1:
                    s_ = s+1
                    Q[s]['R'] = 0
                    T[(s,'R')] = (s_,maze[i][j+1])
                if maze[i][j]!=0:
                    terminals.add(s)
                s+=1
        return Q,T,terminals

    maze = [[0, 0, 0, 0],
            [0, -1, 0, 0],
            [0, -1, -1, 0],
            [0, -0, 0, 1]]

    import pprint
    if __name__ == "__main__":
        Q,T,terminals = init_game_maze(maze)
        Q = Q_Learning(Q,T,terminals,500)
        print('\r\nQ-table:\n')
        pprint.pprint(Q)  #print(Q)
        s = 0
        path = shortest_path(s,Q,T,terminals)
        print("path: ",path)
```


执行上述程序，输出：

```
Q-table:

[{'D': 5.463385240541213e-05, 'R': 0.08605219857171771},
 {'D': -0.9774716004550608, 'L': 0.01417849215406633, 'R': 0.1874107854625826},
 {'D': 5.261481245798102e-05,
  'L': 0.05034373087641624,
  'R': 0.3204861038798131},
 {'D': 0.5317659509722789, 'L': 0.008816349600877973},
 {'D': 5.46520216339789e-05,
  'R': -0.9999999999999996,
  'U': 0.028303397085610464},
 {'D': 0, 'L': 0, 'R': 0, 'U': 0},
 {'D': -0.7458134171671,
  'L': -0.6513215599000001,
  'R': 0.0,
  'U': 0.04289205200013416},
 {'D': 0.749004339861237, 'L': 0.0024304067844035946, 'U': 0.0},
 {'D': 1.4053037959801073e-06,
  'R': -0.94185026299696,
  'U': 0.001731465700844264},
 {'D': 0, 'L': 0, 'R': 0, 'U': 0},
 {'D': 0, 'L': 0, 'R': 0, 'U': 0},
 {'D': 0.94185026299696, 'L': -0.5217031000000001, 'U': 0.10239243331319928},
 {'R': 1.2647734163820967e-07, 'U': 5.46520216339789e-05},
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
 {'L': 2.670077212362204e-06, 'R': 0, 'U': -0.1},
 {'L': 0, 'R': 0, 'U': 0},
 {'L': 0, 'U': 0}]
path: [0, 1, 2, 3, 7, 11, 15]
```

读者可将上述Q-Learning算程序用于求解其他强化学习问题，如用Q-Learning算法玩各种游戏。还可以进一步学习和深度学习结合的Q-Learning算法，如DQN等算法。

4.8 习题

1．下列哪个选项是不正确的？（　　）

A．x = 0b101　　B．x = 0x4f5　　C．x = 19023　　D．x = 03964

2．下列代码输出的结果是什么？（　　）

```
print(1j)
```

A．<type 'float'>　　B．<type 'dict'>　　C．<type 'unicode'>　　D．<type 'complex'>

3．下列说法是否正确？（　　）

Python 有两种数值类型：整数和有符号数。

A．True　　B．False

4．写一段程序，分别以二进制、十进制、八进制、十六进制形式输出整数 23。

5．语句 print(0.2 + 0.2 = = 0.5) 的输出结果是(　　)。

A．True　　B．False　　C．机器依赖　　D．错误 Error

6．下列哪条语句是错误的？（　　）

A．float('inf')　　B．float('nan')　　C．float('56'+'78')　　D．float('12+34')

7．编写一个函数 sin(x)，按照第 2 章的习题第 34 题计算正弦函数 sin(x) 的值, 用 math 模块的函数 pow() 和 factorial() 计算指数和阶乘，调用下列代码测试你的函数 sin(x)，并和 math 模块自带的函数 sin(x) 的结果进行比较。

```
print(sin(math.pi/2))    # returns 1.0
print(sin(math.pi/4))    # returns 0.7071067811865475
```

8．下列哪个命令可创建一个 list 对象？（　　）

A．list1 = list()　　B．list1 = []　　C．list1 = list([1, 2, 3])　　D．上述都可以

9．插入 5，成为 alist 的第 3 个元素，应该用哪个命令?(　　)

A．alist.insert(3, 5)　　B．alist.insert(2, 5)　　C．alist.add(3, 5)　　D．alist.append(3, 5)

10．下列代码的输出结果是什么？（　　）

```
lst=[[1,2],[3,4]]
print(sum(lst,[]))
```

A．[[3],[7]]　　B．[1,2,3,4]　　C．Error　　D．[10]

11．下列代码的输出结果是什么？（　　）

```
x=[[1],[2]]
print(" ".join(list(map(str,x))))
```

A．[1] [2]　　B．[49] [50]　　C．Syntax error　　D．[[1]] [[2]]

12．下列代码的输出结果是什么？（　　）

```
a=[2,5,7]
a.append([17])
a.extend([3,11])
print(a)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

13. 下列代码的输出结果是什么？（　　）

```
a=list((45,)*4)
print((45)*4)
print(a)
```

14. 下列代码的输出结果是什么？（　　）

```
matrix = [[1, 2, 3, 4],
      [4, 5, 6, 7],
      [8, 9, 10, 11],
      [12, 13, 14, 15]]
for i in range(0, 4):
   print(matrix[i][1], end = " ")
```

15. 下列代码的输出结果是什么？（　　）

```
points = [[1, 2], [3, 1.5], [0.5, 0.5]]
points.sort()
print(points)
```

16. 下列代码的输出结果是什么？（　　）

```
def change(var, lst):
   var = 1
   lst[0] = 44
k = 3
a = [1, 2, 3]
change(k, a)
print(k)
print(a)
```

A. 3
[44, 2, 3]

B. 1
[1,2,3]

C. 3
[1,2,3]

D. 1
[44,2,3]

17. 下列代码的输出结果是什么？（　　）

```
L=["hello", "yes!", "a123"]
[n for n in L if n.isalpha() or n.isdigit()]
```

A. ['hello', 'yes', '123']　　B. ['hello']　　C. ['hello', 'a123']　　D. ['yes!', 'a123']

18. 下列代码的输出结果是什么？（　　）

```
[j for i in range(2,5) for j in range(i*2, 10, i)]
```

19. 编写程序：用 list 表示两个矩阵，计算它们的乘积并放入另一个 list 对象中。

20. 以下哪个答案能够打印出如下的结果？（　　）

```
C:\Hwdong\String.doc
```

A.
```
str='C:\Hwdong\Stri\ng.doc'
print(str)
```

B.
```
str='''C:\\Hwdong\string.doc'''
print(str)
```

C.
```
str='C:\\Hwdo\ng\St\ring.doc'
print(str)
```

D.
```
str='C:\\Hwdong\String.doc'
print(str)
```

21. 下列语句的输出结果是什么？（　　）

```
print("\x48\x49!")
```

A. \x48\x49!　　B. 4849　　C. 4849!　　D. 48　　E. 49!　　F. HI!

22. 下列代码的输出结果是什么？（　　）

```
str='hello world'
print (str[::-1])
```

A. world　　B. dlrow olleh　　C. hello world　　D. hello

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

23．下列代码的输出结果是什么？（　　）

```
str='Python is easy to learn, C is hard'
print(str.rfind('s'))
print(str.rfind('r'))
L = len(str)
print(str.count('is', -(L-2), -1))
print(str.endswith("rd"))
```

24．Web 网页中的标记都是以"<>"开头和结尾的，中间的部分就是标记的内容。

例如，string2表示的是一个超链接。编写代码抽取其中的内容(string2)。

25．下列语句的输出结果是什么？（　　）

```
print("Hello {name1} and {name2}".format('foo', 'bin'))
```

A．Hello foo and bin　　B．Hello {name1} and {name2}

C．Error　　D．Hello and

26．下列程序的输出结果是什么？（　　）

```
print('The sum of {0:b} and {1:x} is {2:o}'.format(2, 10, 12))
```

A．The sum of 2 and 10 is 12　　B．The sum of 10 and a is 14

C．The sum of 10 and a is c　　D．Error

27．下列代码的输出结果是什么？（　　）

```
print('abcdefcdghcd'.split('cd', 2))
```

A．['ab', 'ef', 'ghcd']　　B．['ab', 'efcdghcd']

C．['abcdef', 'ghcd']　　D．none of the mentioned

28．编写程序，输入一个字符串，返回一个由首尾各两个字符组成的字符串。例如，string 返回的字符串是 stng；如果原字符串长度小于 2，则返回一个空串。

29．编写程序，从键盘输入多个用逗号隔开的字符串，将它们拼接成一个字符串。例如，输入：

```
'hello','world','abc'产生的是'hello world abc'。
```

30．假如要将 tuple 对象 test 的第 3 个元素修改为“Python”，下面哪条语句是正确的？（　　）

A．test[2] = 'Python'　　B．test(2) = 'Python'

C．test[3] = 'Python'　　D．不能修改 tuple 的元素

31．下列代码输出结果是什么？（　　）

```
my_tuple = (1, 2, 3, 4)
my_tuple.append( (5, 6, 7) )
print(len(my_tuple))
```

A．5　　B．7　　C．2　　D．Error

32．下列代码的输出结果是什么？（　　）

```
d = {}
d[(1,2,4)] = 8
d[(4,2,1)] = 10
d[(1,2)] = 12
sum = 0
for k in d:
    sum += d[k]
print (len(d) + sum)
```

A．30　　B．24　　C．33　　D．12

33．下列代码的输出结果是什么？（　　）

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
a = ('hello',)
n = 2
for i in range(int(n)):
    a = (a,)
    print(a)
```

A．错，tuples 是不可修改的

B．((‘hello’,),)
(((‘hello’,),),).

C．((‘hello’,)’hello’,)

D．((‘hello’,)’hello’,)
(((‘hello’,)’hello’,)’hello’,)

34．下列代码的输出结果是什么？（　　）

```
a,b=6,7
a,b=b,a
a,b
```

A．(6,7)　　B．Invalid syntax　　C．(7,6)　　D．没有任何输出

35．下列代码的输出结果是什么？（　　）

```
a=(1,2)
b=(3,4)
print(a+b)
```


A．(4,6)

B．(1,2,3,4)

C．错，因为 tuple 是不可修改的

D．上述说法都不对

36．关于 set，正确的说法是（　　）。

A．一个 set 是数据元素的无序集合

B．set 中的元素是唯一的

C．不同于 tuple，可以修改 set 中的元素

D．上述说法都对

37．下列哪个创建 set 对象的语句是错误的？（　　）

A．set([[1,2],[3,4]])

B．set([1,2,2,3,4])

C．set((1,2,3,4))

D．{1,2,3,4}

38．下列代码输出结果是什么？（　　）

```
a={4,5,6}
b={2,8,6}
a+b
```

A．{4,5,6,2,8}

B．{4,5,6,2,8,6}

C．Error 错误，因为 set 对象无法使用+运算符

D．Error 错误，因为出现了重复的 6

39．下列哪一条语句产生的是集合{x,y}的对称差？（　　）

A．x | y　　B．x ^ y　　C．x & y　　D．x – y

40．下列代码的输出结果是什么？（　　）

```
a=[1, 4, 3, 5, 2]
b=[3, 1, 5, 2, 4]
print(a==b)
set(a)==set(b)
```

A．True
　False

B．False
　False

C．False
　True

D．True
　True

41．下列代码的输出结果是（　　）。

```
a={3,4,5}
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
a.update([1,2,3])
a
```

A．错，set 类型没有 update()方法　　B．{1, 2, 3, 4, 5}

C．错, list 不能加到 set 里　　D．错, list 中有重复元素

42．下列代码的输出结果是什么？(　　)

```
d = {0: 'a', 1: 'b', 2: 'c'}
for x, y in d:
    print(x, y)
```

A．0 1 2　　B．a b c　　C．0 a 1 b 2 c　　D．上述说法都不对

43．下列代码的输出结果是什么？(　　)

```
d = {0: 'a', 1: 'b', 2: 'c'}
for x, y in d.items( ):
    print(x, y)
```

A．0 1 2　　B．a b c　　C．0 a 1 b 2 c　　D．上述说法都不对

44．下列代码的输出结果是什么？(　　)

```
squares = {1:1, 2:4, 3:9, 4:16, 5:25}
print(squares.pop(4))
print(squares)
```

A．16
{1: 1, 2: 4, 3: 9, 5: 25}

B．16
{1: 1, 2: 4, 3: 9, 4:16, 5: 25}

C．4
{1: 1, 2: 4, 3: 9, 5: 25}

D．4
{1: 1, 2: 4, 3: 9, 4:16, 5: 25}

45．下列代码的输出结果是什么？(　　)

```
a={1:"A",2:"B",3:"C"}
b={4:"D",5:"E"}
a.update(b)
print(a)
```

A．{1: ‘A’, 2: ‘B’, 3: ‘C’}　　B．dict 没有 update()方法

C．{1: ‘A’, 2: ‘B’, 3: ‘C’, 4: ‘D’, 5: ‘E’}　　D．{4: ‘D’, 5: ‘E’}

46．下列代码的输出结果是什么？(　　)

```
a={}
a[2]=1
a[1]=[2,3,4]
print(a[1][1])
```

A．[2,3,4]　　B．3　　C．2　　D．An exception is thrown

47．编写程序，将如下的多个字典：

```
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
```

合并为 {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}。

48．网上有一篇 Painless Q-Learning 的文章介绍了一个机器人从一个建筑物中的任意一个房间走出建筑物的探索问题，如图 4-4(a)所示是建筑物的房间布局。

将房间作为顶点，将两个房间之间的门作为边，可以构造如图 4-4(b)所示的图，图中边上的权值表示从一个门走向另一个门的奖励。针对这个问题，请修改函数 build_T_table()，用于初始化该问题的数据，然后运行函数 main()，观察执行结果。注意：目标状态走向自身所在的边在程序中可以忽略。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

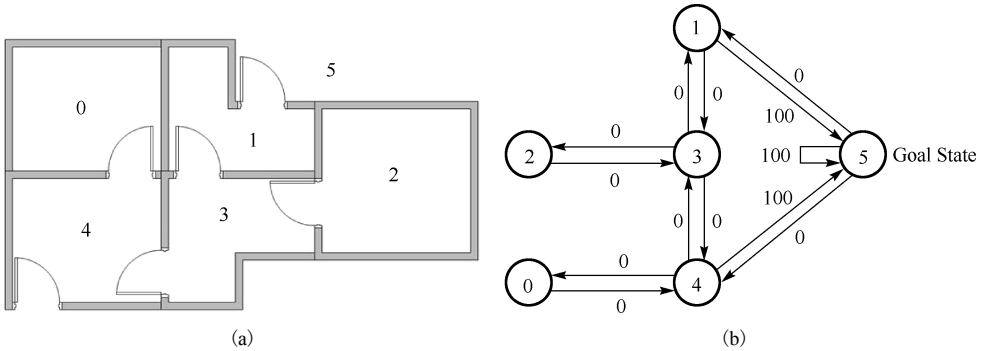


图 4-4　走出房间问题

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 5 章　面向对象编程

5.1　什么是面向对象编程

5.1.1　过程式编程和面向对象编程

程序是由数据和对数据处理的指令(语句)组成的。传统的**过程式编程**用文字量和变量(对象)表示数据，用函数(过程)对这些数据进行处理。对于简单的问题，程序可以用一个函数对数据进行处理，对于复杂问题，程序可以通过采用“分而治之”的思想，将一个大问题分解为一些更小的问题，对这些小问题再分别用单独的**程序块**(称为**过程**或**函数**)进行处理。例如，在前面的游戏程序中，除游戏的主函数外，还有一些初始化数据、处理事件、更新游戏状态(数据)、绘制场景等负责专门功能的函数(过程)。即程序中一个函数中可能会调用其他函数，函数之间通过这种相互调用，协作完成一个复杂问题的程序设计任务。

在过程式编程中，数据和处理数据的过程(函数)是一种松散的关系，也就是说，同样的数据可以被程序中的所有过程(函数)访问，而一个函数也可以访问程序中的不同数据。

过程式编程通过这种“分而治之”的解决问题的方法处理复杂的问题，使程序结构清晰，从而提高了程序开发效率，增加了程序的可靠性、可读性和代码的复用性。然而，程序中的数据都是分散的，任何代码都可以访问这些数据，如果数据出现了异常，则需要在整个软件系统中查找导致错误的处理代码，使得程序难以有效地跟踪维护和调试。另外，这些数据都是一些内在数据类型的对象，少量底层的内在数据类型不利于表示实际问题中的各种丰富概念，如游戏中的各种不同的精灵、员工管理系统的员工、电子商务中的订单等概念。

与过程式编程不同，**面向对象编程**(Object Oriented Programming，OOP)模拟人类思考问题的方法，将一个软件系统看作由一个个具体对象构成，每个对象不仅包含这个对象自身的所有信息，还具有自己特有的功能。例如，游戏中的每个精灵不仅具有位置、大小、图像、生命值等数据属性，还具有运动、更新自身状态等行为能力。面向对象的系统中，对象之间通过收发消息协作完成相关的任务。例如，一个员工向另外一个员工传达一个通知，接到通知的员工就会执行某个动作或行为。

面向对象编程通常分为三个步骤。

首先，需要对系统进行分析，识别系统中有哪些种类的对象，即思考系统中有哪些概念。例如，开发 Pong 游戏程序时，需要分析其中有游戏窗口、画面背景、挡板、球等不同的对象。又如，电子商务系统中有商家、消费者、各种商品、订单、购物车、物流信息等各种概念的对象。除识别出各种概念外，面向对象编程还要分析这些概念之间的关系。例如，一个员工管理系统中有员工、部门，员工中又分为经理、财务、销售等不同类型的员工。员工和经理概念之间有一种**一般到特殊**的关系，即经理是一个特殊的员工，经理不仅具有一般员工的信息和功能，而且还有一些自身特有的信息和功能，如经理具有一定的级别，经理能管理一组员工。同样，部门和员工、经理之间具有一种**包含**的关系，即一个部门可能包含多个员工和经理。

其次，对于每个概念还要分析这个概念的所有对象具有哪些共同的属性。属性包括描述对象状态的**数据属性**和描述对象行为能力的**功能属性**。面向对象编程用类描述其所包含的所有对象的共同特性(属性)，即数据属性(也称**数据成员**或**成员变量**)和功能属性(也称**成员函数**)，但为了与普通的全局函数进行区分，类的成员函数通常被称为“**方法**”。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

最后，需要确定系统中应该有哪些具体的属于不同类的对象，以及这些对象如何发送和接收消息来协作完成一个任务，并根据类来创建这些对象。

5.1.2 Python 既支持面向对象编程，也支持过程式编程

Python 是一种面向对象的编程语言，几乎 Python 中所有的内容都是一个对象，并具有数据的方法和属性。在 Python 中，一个类的对象也称为这个类的一个**实例**。例如，hello 就是一个字符串类 str 的对象：

```
type("hello")
```

输出：

```
str
```

字符串类 str 是刻画所有字符串对象的一个蓝图(蓝本)，而 hello 是 str 类的一个具体**实例**(也称**对象**)。可以利用内置函数 dir()查询字符串类 str 的属性。例如：

```
dir('hello')   # 或者 dir(str)
```

输出：

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace','rfind','rindex','rjust','rpartition','rsplit','rstrip','split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

所谓向一个类对象发送消息，就是通过该对象和**成员访问运算符.**调用类的方法，进行相应的计算或处理。其格式是：

```
类对象.方法名(实参)
```

例如，给字符串类 str 的对象 hello 发送一个 upper()方法消息：

```
"hello".upper()
```

输出：

```
'HELLO'
```

字符串类 str 的 upper()方法会将 hello 中的所有字母转化为大写字母，并构造一个新的 str 对象作为返回结果。

而过程式编程则是用一个单独的函数对数据进行处理。其格式是：

```
函数名(实参)
```

例如，利用函数 len()计算字符串对象 hello 的长度(字符个数)：

```
len("hello")
```

输出：

```
5
```

Python 既支持过程式编程，也支持面向对象式编程，且两种编程方式可以混合使用。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

当然，字符串类 str 等内在类是 Python 已经定义好的类，程序员还可以定义自己的类。在使用时，用户自定义的类和内在类是没有任何区别的。因此，Python 的一切数据都是对象，都是某个类的具体实例，都有相应的数据属性和方法属性。

5.1.3 打印员工信息

以编写一个打印员工信息的程序为例，说明“过程式编程”和“面向对象编程”的程序设计方法之间的区别。

假设一名员工的信息包含姓名、薪水。

按照“**过程式编程**”方法，可以用两个变量 name 和 salary 来表示一名员工的姓名和薪水，然后编写一个具体的过程(或函数)完成打印功能：

```
def printInfo(name, salary):
   print(name,",",salary)

#假设程序里有两名员工，可以表示如下
e1_name = "李平"
e1_salary = 7360.5
e2_name = "张伟"
e2_salary = 5120.8

#打印输出他们的信息
printInfo(e1_name,e1_salary)
printInfo(e2_name,e2_salary)
```


输出：

```
李平 , 7360.5
张伟 , 5120.8
```

按照“**面向对象编程**”方法，首先用关键字 class 来定义一个类，如用类 Employee 表示员工这个抽象概念：

```
class Employee:
      pass
```

以上是用关键字 class 后面跟一个类名 Employee。和用 def 定义一个函数类似，类名后要有一个冒号:。和定义函数一样，类定义中也要描述类的属性(数据属性和功能属性，即方法)。这里暂时用一个空语句 pass 作为占位符，表示暂时空着，后面可以逐步添加相关属性。

有了类，就可以定义这个类的对象：

```
e = Employee()
```

类似于函数调用，类名后跟一对圆括号()，这样就创建了一个类对象，然后给这个类对象一个变量名 e，即通过 e 引用这个类对象。

用函数 dir()查看这个类对象有哪些属性：

```
dir(e)
```

输出：

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
```

类 Employee 并没有定义任何属性，但是为什么 e 会有这么多属性呢？这是因为 Python 的类默

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

认继承了类 object。这些属性实际是从类 object 继承下来的。可以用函数 isinstance()检查一个对象是否为一个类的对象(实例)：

```
isinstance(e,object)
```

输出：

```
True
```

结果 True 说明类 Employee 的对象 e 也是类 object 的对象。可以用函数 issubclass()检查一个类是否是另外类的子类：

```
issubclass(Employee,object)
```

输出：

```
True
```

结果 True 说明类 Employee 是类 object 的一个**子类**，或者说，类 Employee 是继承类 object 的一个**派生类**。

可以通过成员访问运算符.给一个类的对象添加属性，如添加姓名(name)和薪水(salary)数据属性(成员变量)：

```
e.name = 'Li Ping'
e.salary  = 5000
```


再通过函数 dir()检查它的属性：

```
dir(e)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'name', 'salary']
```

可以看到，这个对象确实多了两个属性，name 和 salary。

下面修改类 Employee 的定义，直接在其中添加一个方法 printInfo()，用于打印一个类 Employee 的 name 和 salary 信息：

```
class Employee:
   def printInfo(self):
      print(self.name,",",self.salary)
```

注意

这个方法中有一个叫作 self 的特殊参数，该参数用于指向(引用)将来调用这个方法的那个对象，self.name 和 self.salary 就是 self 引用的对象的 name 和 salary。

然后，创建这个类的两个对象 e1 和 e2。

```
e1 = Employee()
e2 = Employee()
#通过成员访问运算符.访问其数据成员和成员函数(方法)
e1.name = "李平"
e1.salary = 5000
e2.name = "张伟"
e2.salary = 6000

e1.printInfo()    # 向 e1 发送一条消息: printInfo()
e2.printInfo()    # 向 e2 发送一条消息: printInfo()
```

输出：

```
李平 , 5000
张伟 , 6000
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

通过成员访问运算符.向对象 e1 和 e2 发送消息 e1.printInfo() 和 e2.printInfo()，以输出 e1 和 e2 的信息。

总结

- 面向对象编程就是用类刻画同类对象的共同特性，通过类的实例来表示具体的对象，并通过向对象发送消息，即调用类的方法来请求对象进行数据处理或计算。
- 过程式编程将问题分解为一些过程或函数，通过向函数传递数据执行数据处理或计算。
- Python 既支持过程式编程，也支持面向对象式编程，两种编程方式可以混合使用。
- 用关键字 class 定义一个类，类的属性有数据属性和方法属性。习惯上将数据属性称为成员变量，将方法属性称为方法，一个类的实例也称为对象，可以通过一个对象和成员访问运算符.访问这个对象的属性。

5.2 类和对象

类(class)是对一个抽象概念的描述，它描述了属于同一个概念的所有对象的共同属性，这些属性包括数据属性和方法属性。数据属性描述了该类对象的状态，而方法属性描述了该类对象具有哪些功能。

类(class)是刻画该类所有对象共同特性的蓝图(模具)，而对象(object)则是这个类的一个具体实例。例如，Python 的字符串类 str 描述了所有字符串对象的共同属性，而一个具体的字符串对象如"hello"就是这个类的一个具体实例。

面向对象编程的主要工作如下。

- 识别要解决的问题或软件系统中的概念，并用类来描述这些概念。对于一个具体类，可描述其应该有哪些数据属性和方法属性。
- 分析不同的类描述的概念之间具有哪些关系，如是**包含**还是**继承**关系。例如，一辆车包含了车轮、引擎、驾驶室等，这属于包含关系。而继承关系则表达了一个概念是否继承另外一个概念的属性。例如，“经理”作为一个特殊的“雇员”，他除继承了一般雇员的属性外，还有自身特有的属性。包含关系是通过类的数据属性表示的，而继承关系是通过定义派生类的方式表示的。
- 确定系统中有哪些对象，以及这些对象是如何通过发送和接收消息协作完成一个任务的。例如，一个公司可能有几个经理和很多普通雇员，他们共同协作完成某个项目或保证公司的正常运转。

5.2.1 定义类

Python 用关键字 class 定义一个类。例如，定义一个叫作 Employee 的类：

```
class Employee:
    pass
```

定义一个类的格式如下：

```
class 类名:
    类体
```

可以看到，定义一个类的方法是以关键字 class 开头，然后是类名，类名后是冒号:，接着是刻画类的属性的类体。上例的类 Employee 用 pass 空语句作为一个占位符，表示这是一个空的类体，这是为了保证语法的正确性。实际定义类时，将会用实际代码取代这个占位符语句 pass。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

有了类的定义，就可以定义类的实例，即类的对象：

```
e = Employee()
type(e)
```

输出：

```
__main__.Employee
```

可以看到，类名 Employee 后面跟一对圆括号，即 Employee()定义(创建)了一个类的对象，创建一个类的对象也称“**类的实例化**”，这个对象也称“实例”。

变量名 e 引用这个创建的 Employee 类对象(实例)。通过调用函数 type()验证了 e 指向的对象的数据类型确实是 Employee 这个类。

前面讲过，Python 的类都默认继承自类 object，即其父类是 object。因此，该类是其父类 object 的子类。例如：

```
issubclass(Employee,object)
```

输出：

```
True
```

5.2.2 实例属性和构造函数

创建一个类对象(实例)后，可以直接用成员访问运算符.给这个实例添加不同的属性。例如：

```
e = Employee()
e2 = Employee()
e.name = 'LiPing'
e.salary = 5000
e2.name = 'WangWei'
e2.salary = 6000
print(e.name,e.salary)
print(e2.name,e2.salary)
```

输出：

```
LiPing 5000
WangWei 6000
```

Python 创建一个类的实例(对象)是通过一个叫作**构造函数**的__init__()方法完成的。类 Employee 虽然没有定义这个构造函数，但 Python 会自动生成一个默认的__init__()方法。例如：

```
class Employee:
    def __init__(self):
        super().__init__()
```

与普通函数类似，这个默认的构造函数也是用关键字 def 来定义的，并且其第一个参数必须是叫作 self 的参数，这个参数指向(引用)要创建的对象。

默认的构造函数会默认调用其父类的 super().__init__()方法，并对这个对象从父类继承下来的属性进行初始化。super()用于得到这个类的父类。当用 Employee()创建一个类的对象时，会自动调用这个构造函数对创建的对象进行初始化。为验证这一点，可以在类 Employee 的这个方法中人为地添加一条打印语句。例如：

```
class Employee:
    def __init__(self):
        print('Employee 构造函数用于创建一个对象')
        super().__init__()
e = Employee()
```

输出：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    Employee 构造函数用于创建一个对象
```

类的构造函数__init__()方法除 self 参数外，还可以传递其他参数，通常传递用于初始化实例属性的参数。例如：

```
class Employee:
   def __init__(self, Name, Salary):
      self.name = Name
      self.salary = Salary
```

这个构造函数为创建的对象(实例)添加了叫作 name 和 salary 的属性变量，并用传入的参数 Name 和 Salary 对它们初始化。这种针对具体实例(对象)的属性，如 self.name 和 self.salary，称为这个对象的“**实例属性**”。

现在，可以创建一个 Employee 对象：

```
e = Employee('Li ping',5000)
print(e.name,' \t',e.salary)
```

输出：

```
Li ping       5000
```

通过 Employee('Liping',5000)创建对象时，会自动调用__init__()构造函数。

构造函数有三个参数，但在创建对象时，只要传递除 self 外的其他参数就可以了，不需要也无法传递 self 参数。当然，如果传入的实参个数少于两个或多于两个，则会出错。例如：

```
e = Employee('Li ping')
```

产生 TypeError(类型错误)“__init__()缺少一个需要的位置参数：'Salary'”：

```
---------------------------------------------------------------------------

TypeError                       Traceback (most recent call last)

<ipython-input-17-b342e3174e65> in <module>()
----> 1 e = Employee('Li ping')
TypeError: __init__() missing 1 required positional argument: 'Salary'
```

同样，执行：

```
e = Employee('Liping',5000,'hi')
```

导致类似的错误：

```
---------------------------------------------------------------------------

TypeError                       Traceback (most recent call last)

<ipython-input-18-3151b6cb5786> in <module>()
----> 1 e = Employee('Liping',5000,'hi')
TypeError: __init__() takes 3 positional arguments but 4 were given
```

当然，构造函数的参数名和对象实例属性名相同也是可以的。例如：

```
class Employee:
   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
```

再看下面的代码：

```
e = Employee('Li ping',5000)
e2 = Employee('Wang Wei',6000)
print(e.name,' \t',e.salary)
print(e2.name,' \t',e2.salary)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
Li ping       5000
Wang Wei      6000
```

可以看到，上例创建了两个 Employee 对象，每个对象具有不同的实例属性值。

注意

实例属性是属于具体的实例的，只能通过**实例名.实例属性**来访问实例属性，不能通过**类名.实例属性**来访问实例属性。例如:

```
Employee.name
```

上述语句将抛出“AttributeError”错误:

```
---------------------------------------------------------------------------
AttributeError                Traceback (most recent call last)
<ipython-input-11-30b1d128b34e> in <module>()
----> 1 Employee.name

AttributeError: type object 'Employee' has no attribute 'name'
```

5.2.3 实例方法

除构造函数外，还可以给类添加更多的方法(成员函数)。例如，添加一个 printInfo()的方法：

```
class Employee:
   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
   def printInfo(self):
      print(self.name,",",self.salary)
```

该方法同样有一个 self 参数，用于指向(引用)调用这个方法的具体对象。然后通过：

```
print(self.name,",",self.salary)
```

输出这个 self 引用的对象的 name 和 salary 属性变量。

下列代码通过 Employee 对象调用这个.printInfo()方法：

```
e = Employee('Liping',5000)
e.printInfo()      #通过 e 调用 Employee 的 printInfo()
e2 = Employee('Wang wei',6000)
e2.printInfo()     #通过 e2 调用 Employee 的 printInfo()
```

输出：

```
Liping , 5000
Wang wei , 6000
```

可以看到，类的 printInfo()方法可以打印 self 引用的那个调用该方法的对象的信息。两个对象虽然都有 name 和 salary 属性，但它们的值是不一样的。

因此，将 printInfo()方法这种针对不同类的实例(对象)产生不同计算结果的方法称为**实例方法**。

再如，定义一些查询和修改数据属性的方法：

```
class Employee:
   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
   def printInfo(self):
      print(self.name,",",self.salary)
   def set_name(self,name):
      self.name = name
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    def get_name(self):
        return self.name
    def set_salary(self,salary):
        self.salary = salary
    def get_salary(self):
        return self.salary
```

注意

所有实例方法的第一个参数都必须是 self，即引用调用这个实例方法的那个对象。

例如，测试上述的实例方法：

```
e = Employee('Li ping',5000)
e.printInfo()
e.set_name('Wang Wei')    #通过方法 set_name 修改 e 的 name 属性
e.set_salary(5500)        #通过方法 set_salary 修改 e 的 salary 属性
print(e.get_name(),'\t',e.get_salary())
e.printInfo()
```

输出：

```
Li ping , 5000
Wang Wei , 5500
Wang Wei , 5500
```

注意

和其他编程语言(如 C++)不同，在同一个类中不能定义多个同名但形参不同的成员函数，即不能定义重载(overloading)成员函数。例如：

```
class X:
    def f(self):
        print("f()")

    def f(self,n):
        print("f(int n)")
```

该类 X 中定义了两个同名的方法 f()，在 Python 中这是不允许的。

5.2.4 类属性

每个对象都有自己单独的实例属性，改变一个对象的实例属性不会影响其他对象的实例属性。除实例属性外，还可以给一个类定义类属性，**类属性**是指类的所有对象都共享的属性，是定义在类的方法外面的属性。

例如，给类 Employee 添加一个类属性 count，用于记录从该类创建的类的实例的个数，每当用构造函数创建一个实例时，作为该类属性的计数器 count 就增加 1：

```
class Employee:
    count = 0
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.count +=1     #或 type(self).count += 1
    def printInfo(self):
        print('Employee 总数: ',self.count)
        print(self.name,",",self.salary)
```

下面的代码创建了两个对象，并分别通过这两个对象和类名访问这个类属性 count：

```
e = Employee('Li ping',5000)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
e2 = Employee('Wang wei',6000)
print(e.count,end = ' ')
print(e2.count,end = ' ')
print(Employee.count,end = ' ')
print()
e.printInfo()
e2.printInfo()
```

输出：

```
2 2 2
Employee 总数： 2
Li ping , 5000
Employee 总数： 2
Wang wei , 6000
```

和实例属性不同，类属性是所有类的实例(对象)共享的，而不是属于每个实例的。可以通过实例的属性__dict__查看实例的实例属性有哪些。例如，

```
print(e.__dict__)
print(e2.__dict__)
{'name': 'Li ping', 'salary': 5000}
{'name': 'Wang wei', 'salary': 6000}
```


由此可见，类属性是不属于一个具体的类实例(对象)的。通常，可以通过**类名.类属性**来查询或修改类属性，也可通过**实例名.类属性**(包括 **self.类属性**)来查询实例属性，但不能通过**实例名.类属性**(包括 **self.类属性**)的方式来修改类属性，否则就是创建了实例属性，而不是访问类属性。例如：

```
class C:
    count = 0;
    def __init__(self):
        C.count +=1
    def inc(self):
        self.count+=1   #等价于 self.count = self.count+1，实际为该实例创建了一个属性 count
c1 = C()
c2 = C()
print(c1.count,c2.count,C.count)  #输出的都是类属性 count
c1.inc()
print(c1.count,c2.count,C.count)  #c1.count 是 c1 的实例属性而不是类属性
print(c1.__dict__)
print(c2.__dict__)
```

输出：

```
2 2 2
3 2 2
{'count': 3}
{}
```

inc()方法中的“self.count+=1”等价于“self.count = self.count+1”，赋值语句的左边 self.count 相当于为这个实例创建该类的一个新的实例属性 count，而右边的 self.count 仅仅是查询，当通过一个实例第一次调用 inc()方法，因为还没有实例属性 self.count，右边的这个 self.count 访问的是类属性，因此，第一次通过实例调用 inc()方法时“self.count+=1”等价于“self.count = Employee.count+1”。再一次通过这个实例调用 inc()方法时，右边的 self.count 就是这个实例已经创建好的实例属性 count，而不是类属性 count 了。例如，可以通过继续执行下述代码来验证这一点：

```
c1.inc()   #self.count = self.count+1 的 self.count 是这个实例已存在的实例属性 count
print(c1.count) #输出值是 4 而不是 3
```

输出：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
4
```

上例代码中的 c2.__dict__并没有实例属性 count，因此，c2.count 就是 C.count。

类属性也可用于给类的实例对象提供默认值。例如，定义如下的类 Date：

```
class Date:
    default_date = [2018,1,1]    #类属性 default_date
    def __init__(self, year=default_date[0],month=default_date[1],day=default_date[2]):
        self.year = year
        self.month = month
        self.day = day
    def printInfo(self):
        print(self.year,'-',self.month,"-",self.day)
```

创建类 Date 的对象可以传递三个代表年(year)、月(month)、日(day)的参数，构造函数中对应的形参都有默认值，如果没有提供相应的参数，则使用类属性 default_date 的值作为这些形参的默认值，并对相应的实例属性进行初始化，以保证创建的类 Date 的对象是一个合法有效的日期。例如：

```
d = Date()
d1 = Date(2015)
d2 = Date(2015,2)
d3 = Date(2015,2,8)
d.printInfo()
d1.printInfo()
d2.printInfo()
d3.printInfo()
```

输出：

```
2018 - 1 - 1
2015 - 1 - 1
2015 - 2 - 1
2015 - 2 - 8
```

5.2.5 del

del 运算符可以用于删除一个变量，该变量的引用计数会减少 1，当一个对象的引用计数变为 0 后，Python 系统会自动回收这个对象占用的内存空间。

del 运算符当然也可以用于删除一个对象的实例属性或类属性，因为这些属性实际上也是其他类型的对象的引用。例如：

```
e = Employee('Li ping',5000)
e2 = Employee('Wang wei',6000)
del e.name
print(e.__dict__)
print(e2.__dict__)
```

输出：

```
{'salary': 5000}
{'name': 'Wang wei', 'salary': 6000}
```

用 del 运算符删除了 e 的实例属性 name，但对其他实例(如 e2)的实例属性 name 没有任何影响。同样，del 运算符也可以用于删除类属性。例如：

```
print(Employee.count)
```

输出：

```
2
```

再执行：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
del Employee.count
print(Employee.count)
```

上例访问不存在的(类)属性 count，抛出了 AttributeError(属性错误)“'Employee'没有属性'count'”：

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-46-2ed66471fb2b> in <module>()
      1 del Employee.count
----> 2 print(Employee.count)
AttributeError: type object 'Employee' has no attribute 'count'
```

也可以用 del 运算符删除一个对象(实际是删除对象的引用，当对象的引用计数等于 0 时，该对象才真正被删除)。例如：

```
e3 = Employee('Li ping',5000) #对象创建时引用计数是 1，被 e3 引用，引用计数变为
e4 = e3                       #该对象又被 e4 引用，因此这个对象的引用计数为 2.
['Employee', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'e3', 'e4']
del e3                        #这个对象的引用计数为 1
```


del e3 删除了 e3，对象的引用计数减少 1，e4 引用的对象的引用计数是 1，所以该对象是存在的。

```
dir()
```

输出：

```
['Employee', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'e4']
```

由此可见，e3 已被删除，再继续删除 e4：

```
del e4
dir()
```

输出：

```
['Employee', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__']
```

此时，e4 也被删除了，同时，它引用的那个对象也会被删除。

5.2.6 访问控制和私有属性

面向对象编程的思想是通过类定义描述一个概念，从而将一个类对象的属性都封装在一个对象中，并且还通过信息隐蔽将一个对象的数据(状态)保护起来，以防止一个对象的状态被外界无意或有意篡改、窃取或破坏。面向对象编程语言可提供访问控制，用于控制外界能够访问一个对象的哪些属性。通常，表示对象的数据属性大多是隐藏的，即外界无法直接访问，但外界可以通过类自身的一些公开的方法访问类对象的属性。

例如，上例的类 Employee 的对象的数据属性是可以被外界任意访问修改的：

```
e.name = "李萍"
e2.salary = 7800
e.printInfo()
e2.printInfo()
```

输出：

```
Employee 总数： 4
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
李萍 , 5000
Employee 总数:  4
Wang wei , 7800
```

Python 可以通过给类的属性名前添加两个下画线__来禁止外界访问这些属性，加了这两个下画线后属性就成为了私有属性。例如，给其中的 name 和 salary 属性名前添加两个下画线__，它们就成为“**私有变量**”，外界就不能直接访问这些属性(变量)了：

```
Class Employee:
    '这是一个描述公司普通雇员的类'
    def __init__(self,name,salary):
        self.__name = name
        self.__salary = salary
    def printInfo(self):
        print(self.__name,",",self.__salary)

e = Employee("李平",5000)
e2 = Employee("张伟",5000)
e.printInfo()
e2.printInfo()

e.__salary = 7000   #试图直接访问 e 的数据变量
e.printInfo()
print(e.__salary)
```


输出：

```
李平 , 5000
张伟 , 5000
李平 , 5000
7000
```

尽管执行了“e.__salary = 7000”，但 e.printInfo()的输出信息显示的 salary 仍然是 5000，这就说明不能直接访问修改对象的私有变量。然而，最后一句的“print(e.__salary)”的输出信息为什么又显示 7000 呢？这是因为，Python 可以随时给一个实例绑定实例变量。实际上，e 除私有变量外，又多了一个变量__salary，原来的私有变量实际上被 Python 修改成了_Employee__salary，即在这些名字前加上了一个带下画线的类名。

```
Print(e._Employee__salary)
```

输出：

```
5000
```

因此，在 Python 中，实际上是无法真正做到访问控制的，实际上外界仍然是可以访问私有属性的。例如，外界不能直接访问__salary，但还有可以通过修改的名字 e._Employee__salary 去访问并修改它：

```
e._Employee__salary  = 3000
e.printInfo()
print(e._Employee__salary)
```

输出：

```
李平 , 3000
3000
```

因此，和其他编程语言不同，Python 不能真正做到访问控制。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

5.2.7 运算符重载

使用人们熟悉的运算符对数据进行运算，要比使用函数对数据进行运算更加直观且易于理解，如表达式“2+3*5”，如果用函数可能写成“add(2,multi(3,5))”，显然前者比后者更加一目了然。

对某种类型的对象要使用某种运算符，如加法运算符+，就必须对这种类型重新定义相应的运算符函数，Python 对于 int 整型、float 浮点型、str 字符串类型等都重新定义了加法运算符函数，从而可以直接使用加法运算符对这些类型的对象进行运算。

对一个类型重新定义运算符函数的行为称为“**运算符重载**”。例如，下例的 Vector2 是一个表示数学中的二维几何向量的类，其中重载了加法运算符对应的运算符方法，即**__add__**()：

```
class Vector2:
    def __init__(self,x,y):
        self.__x = x
        self.__y = y
    def __add__(self,other_v):
        return Vector2(self.__x+other_v.__x, self.__y+other_v.__y)
                                                    #返回一个新的 Vector2 对象
    def print(self):
        print(self.__x,self.__y)
```


现在，可以使用加法运算符对两个 Vector2 的对象进行加法运算：

```
v = Vector2(2.5,3.5)
v2 = Vector2(10.5,30.5)
v3 = v+v2   #等价于 v.__add__(v2)，即实际调用的是方法__add__()
v3.print()
13.0 34.0
```

还可以重载其他运算符，如比较运算符(如<、>、==、!=)等。例如，下列代码重载了“等于运算符函数”，即__eq__()方法。

```
class Vector2:
    #...
    def __eq__(self,other_v):
        return self.__x==other_v.__x and  self.__y==other_v.__y
```

现在，就可以用==运算符对两个 Vector2 对象进行比较，查看它们的值是否相等：

```
print(v==v2)  #“v==v2”等价于 v.__eq__(v2)
```

输出：

```
False
```

表 5-1 是常见的运算符对应的运算符方法，用户在定义自己的类时，可以根据需要重载这些运算符方法。

表 5-1 常见的运算符对应的运算符方法

运 算 符	运算符方法	含 义	用 法
+	__add__(self, other)	加法	a+b
–	__sub__(self, other)	减法	a–b
*	__mul__(self, other)	乘法	a*b
/	__truediv__(self, other)	浮点除法	a/b
//	__floordiv__(self, other)	整数除法	a//b
%	__mod__(self, other)	求余	a%b
**	__pow__(self, other)	指数	a**b

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

续表

运 算 符	运算符方法	含 义	用 法
<	__lt__(self, other)	小于	a<b
<=	__le__(self, other)	小于等于	a<=b
==	__eq__(self, other)	等于	a==b
!=	__ne__(self, other)	不等于	a!=b
>	__gt__(self, other)	大于	a>b
>=	__ge__(self, other)	大于等于	a>=b
&	__and__(self, other)	位与	a and b
|	__or__(self, other)	位或	a or b
^	__xor__(self, other)	异或	a^b
~	__invert__(self)	位取反	~a
<<	__lshift__(self, other)	左移	a<<b
>>	__rshift__(self, other)	右移	a>>b
[]	__getitem__(self, index)	下标运算符	a[2]
in	__contains__(self, val)	包含于	a in b

总结

- 关键字 class 用于定义一个类。一个类具有**实例属性**和**类属性**，每个对象都有自己的独立的实例属性。不同对象可具有不同的实例属性，而类属性是所有对象共享的。
- 通常，用构造函数__init__()定义和初始化一个对象的实例属性。在创建一个类的对象(实例)时，类的构造函数会自动被调用。其中，第一个参数 self 指向(引用)这个创建的对象。
- 类的实例方法的第一个参数必须是 self，用于指向调用这个方法的那个对象。
- 在实例方法中，可以通过类名或 type(self)访问类属性。
- del 运算符用来删除一个变量指向的对象的引用计数，也可以用来删除对象的属性。当对象本身的引用计数为 0 时，Python 的垃圾回收机制会自动回收这个对象占用的内存资源。
- 运算符重载：通过对一个类定义运算符方法，使这个运算符可以用于这种类的对象。

5.3 派生类

5.3.1 派生类

1.派生类

Python 允许在一个已经存在的类的基础上定义一个新的类，新的类会继承已有类的属性，但也会添加自己特有的一些属性，这个新的类就称为“**派生类**”或“**子类**”，而原有的类称为“**基类**”“**父类**”或“**超类**”。

从一个基类(Base)定义一个新的派生类(Derived)的格式如下：

```
class Derived(Base):
    派生类的类体
```

即在要定义的类名后的圆括号里写入基类名就可以了。如果没有以这种方式显式地说明一个类的基类，则默认其基类是 object。

在一个公司的员工中，有一些称为“经理”的特殊雇员，“经理”不仅具有一般“雇员的属性”，还有作为“经理”特有的一些属性，如经理的级别、管理哪些雇员等。因此，在一个公司员工管理

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

程序中，除定义一个表示一般雇员的类 Employee 外，还需要定义一个表示经理的类 Manager。

一种简单的方法是，定义两个互不相干的类。首先编写一般雇员的类 Employee 的代码，然后通过代码复制、粘贴、修改的方法，在一般雇员的类 Employee 的代码基础上编写经理的类 Manager 的代码：

```
class Employee:
    '这是一个描述公司雇员的类'
    def __init__(self,name,salary):
        self.__name = name
        self.__salary = salary
    def printInfo(self):
        print(self.__name, ",", self.__salary)
    def get_name(self):
        return self.__name;
    def set_name(self,name):
        self.__name = name;
    #...

class Manager:
    '这是一个描述公司经理的类'
    def __init__(self,name,salary,level,employees):
        self.__name = name
        self.__salary = salary
        self.__level = level
        self.__employees  = employees
    def printInfo(self):
        print(self.__name,",",self.__salary,",",self.__level)
        for e in employees:
            print(e.get_name())
    def get_level(self):
        return self.__level;
    def set_level(self,level):
        self.__level = level;
    def get_name(self):
        return self.__name;
    def set_name(self,name):
        self.__name = name;
    #...
```


这种方法有以下两个缺点。

- 代码复制、粘贴不仅容易出错，而且也会导致代码的复用性降低，从而不利于调试维护。例如，类 Manager 仍然要重写同样的 set_name()等代码。如果某处代码需要修改，则其他地方的代码也需要进行相应的调整。
- 两个类没有任何关系。尽管程序员知道这两个类存在某种关系，但在 Python 解释器看来，类 Manager 和类 Employee 二者是相互独立、互不相干的。

更好的办法就是，从类 Employee 定义一个派生类 Manager，从而可以避免复制类 Employee 已有的代码，在类 Manager 中只要添加类 Manager 特有的属性，那些类 Employee 已有的属性就会自动被继承下来。这种方法不仅节省代码编写量，提高代码的复用性和可靠性，而且还可以表示出类 Manager 和类 Employee 之间的继承关系。

只要在定义类 Manager 的类名后的圆括号里写入类 Employee 的类名，类 Manager 就会自动继承类 Employee 已有的属性。例如：

```
class Employee:
    '这是一个描述公司普通雇员的类'
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    def __init__(self,Name,Salary):
        self.__name = Name
        self.__salary = Salary
    def printInfo(self):
        print(self.__name,",",self.__salary)
    def get_name(self):
        return self.__name;
    def set_name(self,name):
        self.__name = name;
    #...

class Manager(Employee):
    pass
```

目前定义的子类 Manager 完全继承了基类 Employee 的所有属性。因此，下面代码能够正常运行：

```
m = Manager("李平",5000)
m2 = Manager("张伟",6000)
m.printInfo()
m2.printInfo()
```

输出：

```
李平 , 5000
张伟 , 6000
```


内置函数 isinstance()可以检查一个对象是否是某个类的实例(对象)：

```
isinstance(m,Manager)
```

输出：

```
True
```

执行：

```
isinstance(m,Employee)
```

输出：

```
True
```

因为类 Manager 是从类 Employee 派生出来的，所以一个类 Manager 对象当然也是一个类 Employee 对象，正如“一辆轿车也是一辆车”一样。

派生类 Manager 的类体里没有写任何代码，因此和类 Employee 具有一样属性，正如类 Employee 自动继承了类 object 的属性一样。

通常情况下，需要给派生类添加派生类特有的属性(数据属性和方法)。下例给类 Manager 添加不同于类 Employee 的特有的数据属性，level(经理级别)和 employees(管理的雇员列表)。其中，构造函数__init__()需要传递四个参数：

```
class Manager(Employee):
    '这是一个描述公司经理的类'
    def __init__(self,name,salary,level,employees):
        Employee.__init__(self,name,salary)   #基类构造函数对基类部分初始化
        self.__level = level
        self.__employees  = employees

    def printInfo(self):
        Employee.printInfo(self)
        print("经理级别: ",self.__level)
        print("管理的员工有:")
        for e in self.__employees:
            print(e.get_name())
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    def get_level(self):
        return self.__level
```

类 Manager 继承了类 Employee 的属性(数据和方法)，还修改了其__init__()构造方法和printInfo()方法，并且还定义了新的 get_level()方法：

```
e = Employee("李平",5000)
e2 = Employee("张伟",6000)

employees = []
employees.append(e)
employees.append(e2)
m = Manager("赵四",7000,2,employees)    #调用了 Manager 自身的构造函数
m.printInfo()                           #调用的是 Manager 的新的 printInfo()方法
print()
print(m.get_name(),"的级别：",m.get_level())      #调用从 Employee 继承的 get_name()
                                        #调用 Manager 特有的 get_level()
```

输出：

```
赵四 , 7000
经理级别： 2
管理的员工有:
李平
张伟

赵四 的级别： 2
```


2．super()方法

在派生类中可以通过 super()方法来调用父类的方法。

```
class Manager(Employee):
    '这是一个描述公司经理的类'
    def __init__(self,name,salary,level,employees):
        super().__init__(self,name,salary)   #基类构造函数对基类部分初始化
        self.__level = level
        self.__employees  = employees

    def printInfo(self):
        super().printInfo(self)
        print("经理级别：",self.__level)
        print("管理的员工有:")
        for e in self.__employees:
            print(e.get_name())
    #...
```

使用 super()方法可避免写基类名。

5.3.2 覆盖

子类通过定义和父类同名的属性可以覆盖父类的数据属性和方法。例如，上节例子中类 Manager 和类 Employee 都有一个同样签名(签名包括函数名和参数列表)的 printInfo()方法，派生类(子类)的 printInfo()方法就覆盖了基类(父类)的 printInfo()方法。再如：

```
class Base:
    cvar = 'hello'
    bcvar = 3
    def f(self):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            var = 2
            print(Base.cvar,var)
        def g(self):
            print('函数 g')

    class Derived(Base):
        cvar = 'derived'
        def f(self):
            var  = 3.14
            print(Base.cvar,Derived.cvar,var)

    d = Derived()
    print(d.bcvar)
    print(d.cvar)
    d.g()
    d.f()
```

输出：

```
    3
    derived
    函数 g
    hello derived 3.14
```


可以看到，派生类对象 d 继承了基类的属性，如数据变量 bcvar 和 g()方法，同时覆盖了基类的数据属性，如 cvar 和 f()方法。

5.3.3 多继承

一个类可以继承多个类的特性，即可以定义一个从多个类派生出来的派生类。例如：

```
    class Base1:
        cvar1 = 'base1'
        def f(self):
            var1 = 1
            print(Base1.cvar1,var1)
        def g(self):
            print('函数 g')

    class Base2:
        cvar2 = 'base2'
        def f(self):
            var2 = 2
            print(Base2.cvar2,var2)

    class MultiDerived(Base1, Base2):
        var = [1,2,3]
```

上述类 Base1 中定义了 f()、g()两个方法，类 Base2 中定义了一个 f()方法，类 MultiDerived 是继承这两个类的派生类。那么，类 Base1 的 f()方法和类 Base2 的 f()方法到底是如何被类 MultiDerived 继承的呢？可以用下面代码来测试一下：

```
    m = MultiDerived()
    m.g()
    m.f()
```

输出：

```
    函数 g
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

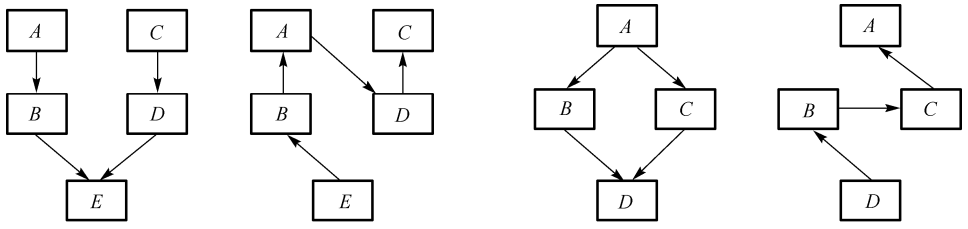
```
base1 1
```

由此说明，类 MultiDerived 继承的是类 Base1 的 f() 方法。

5.3.4 属性解析

虽然派生类继承了类 Base1 和类 Base2 的同样签名的 f() 方法，但是在调用 m.f() 时只调用了类 Base1 的 f() 方法。这是因为，在调用 f() 方法时，Python 解释器首先在该类自身的方法里寻找这个方法，如未找到，就到其上层基类去寻找，在上层基类中又遵循从左到右的顺序查找。这种深度优先、从左到右寻找一个对象的属性的过程称为“**属性解析**”。这种寻找类的属性遵循的解析次序称为“**方法解析次序**”（Method Resolution Order，MRO）。

MRO 采用 C3 算法将具有继承关系的类变成一个线性序列，如图 5-1 所示为两种常见继承关系（并列继承和棱形继承）的 MRO 线形图。



(a) 并列继承关系及其MRO线形图　　(b) 棱形继承关系及其MRO线形图

图 5-1　两种继承关系及其 MRO 线形图

类似图 5-1 (b)，类 MultiDerived 及其基类构成了一个棱形继承关系，因此其 MRO 解析顺序应该是：MultiDerived、Base1、Base2、object。可以通过类的属性__mro__或 MRO() 方法得到一个类的 MRO 顺序。例如：

```
MultiDerived.__mro__
```

输出：

```
(__main__.MultiDerived, __main__.Base1, __main__.Base2, object)
```

再如：

```
class X: pass
class Y: pass
class Z: pass

class A(X,Y): pass
class B(Y,Z): pass

class M(B,A,Z): pass
print(M.mro())
```

类 M 及其基类的继承关系及 MRO 线形图如图 5-2 所示。

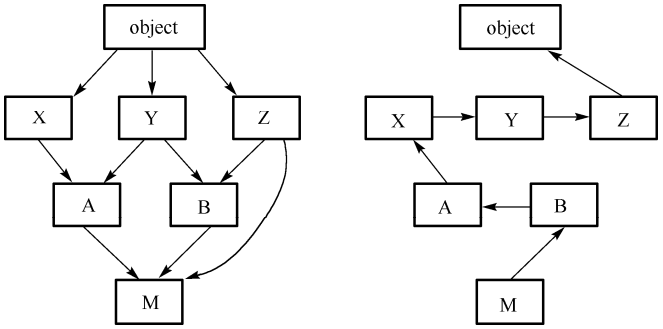


图 5-2　类 M 及其基类的继承关系及 MRO 线形图

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

上述代码输出 MRO 线形序列如下：

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.X'>, <class '__main__.Y'>, <class '__main__.Z'>, <class 'object'>]
```

也可以用 inspect 模块的函数 getmro() 查询一个类的所有基类。例如：

```
import inspect
inspect.getmro(M)
```

输出：

```
(__main__.M,
 __main__.B,
 __main__.A,
 __main__.X,
 __main__.Y,
 __main__.Z,
 object)
```

MRO 确定解析次序的 C3 算法比较复杂，初学者不必纠结于此，只要调用上述函数就可以得到这个 MRO 线形序列。

总结

- 可以从一个已有的类定义一个派生类。派生类也称为“子类”，而其依赖的类称为“父类”“基类”或“超类”。派生类(对象)继承了基类(对象)的属性，同时派生类也可以定义自己特有的属性或覆盖基类的同名属性。派生类的方法可以用 super() 方法调用其基类的方法。
- 派生类可以直接继承多个基类，即从多个基类定义一个派生类，这种定义派生类的方式称为“多重继承”或“多继承”。
- 对一个派生类，可以通过属性解析“深度优先，自左向右”的顺序查找其属性。

5.4 绑定属性

5.4.1 动态绑定：给类和对象任意绑定属性

作为一个动态类型语言，Python 可以随时给一个类及其对象绑定属性，如 5.1.2 节和 5.2.2 节在创建一个类对象后，可以通过成员访问运算符 . 给该对象添加（绑定）数据属性 name 和 salary。例如：

```
class Employee:
   pass

e  = Employee()
e2 = Employee()
e.name = 'Li Ping'
e2.salary = 3000
print(e.__dict__)
print(e2.__dict__)
```

输出：

```
{'name': 'Li Ping'}
{'salary': 3000}
```

上述代码给类 Employee 的两个对象 e 和 e2 绑定了不同的实例属：e.name 和 e2.salary。当然，也可以随时绑定方法属性。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
class Employee:
    pass
def set_name(self,name)
    self.name = name

e = Employee()
e.name = 'Li Ping'
e.set_name = set_name     #给 e 绑定一个 set_name()方法
```

然后通过 e 调用 set_name()：

```
e.set_name("张伟")          #调用 e 的 set_name()方法
```

此时抛出 TypeError 类型错误："TypeError: set_name() missing 1 required positional argument: 'name'"。这是因为，set_name()方法并没有真正被绑定到对象 e 上，即没有真正成为 e 的一个方法属性，因此 set_name()方法就无法知道 self 是什么。

可以通过以下语句验证这一点：

```
isinstance(e.set_name, types.MethodType)
```

输出：

```
False
```


当然，可以将 e 传给这个全局函数 set_name()的第一个参数：

```
e.set_name(e,"张伟")      #调用 e 的 set_name()方法
```

实际上，可以通过 types 模块的 MethodType()方法给一个类或对象绑定方法属性。例如：

```
import types
e.set_name = types.MethodType(set_name,e)  #给 e 绑定一个 set_name 方法
e.set_name("Zhang Wei")                    #调用 e 的 set_name()方法
print(e.name)
isinstance(e.set_name, types.MethodType)
```

输出：

```
Zhang Wei
True
```

由此可见，set_name()方法真正成为了 e 的一个方法属性了。

上面的动态绑定都是将属性绑定到对象 e 上的，Employee 的其他实例（对象）并不具有这些动态绑定的属性。例如：

```
e2 = Employee()
print(hasattr(e,"name"))
print(hasattr(e2,"name"))
print(isinstance(e.set_name, types.MethodType))
print(isinstance(e2.set_name, types.MethodType))
```

输出：

```
True
False
True
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-32-a81c8bc33055> in <module>
     3 print(hasattr(e2,"name"))
     4 print(isinstance(e.set_name, types.MethodType))
----> 5 print(isinstance(e2.set_name, types.MethodType))
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
AttributeError: 'Employee' object has no attribute 'set_name'
```

这说明了对象 e2 没有数据属性 name 和方法属性 set_name()。

当然，也可以给类动态绑定类属性。例如：

```
Employee.type = '雇员'
def SetType(type):
    Employee.type = type

Employee.set_type =   SetType
Employee.set_type('雇员')
print(e.type)
```

输出：

```
雇员
```

上述代码给 Employee 绑定了两个类属性，type 和 set_type。

5.4.2 对象的__dict__属性

这种可以随时给类或对象绑定属性的动态语言的特性非常灵活。但是，如果不同对象具有不同属性则也容易引起混乱。为了管理可能动态增加的属性，Python 采用一个 dict 对象用于管理这些属性，即对于每个对象都有一个单独的__dict__属性，该属性是一个包含了该对象其他所有属性的字典对象。例如：

```
e.__dict__
```

输出：

```
{'name': 'Li Ping',
 'Print': <bound method print_name of <__main__.Employee object at 0x0000016C4998FE10>>}
```

执行：

```
e2.__dict__
```

输出：

```
{'salary': 3000}
```

这个__dict__字典对象一方面要消耗一定的内存，同时，系统管理与维护它时也要消耗一定的时间。为了使一个类的所有对象都具有统一的属性，则可以用__slots__限制绑定的属性，以避免使用__dict__管理所有其他属性而带来时空开销。

5.4.3 __slots__

1. __slots__的用途

通过使用__slots__可以限制一个类能够添加的属性。例如：

```
class Employee:
    __slots__ = ['name','salary','printInfo']

e = Employee()
e.name = 'Li Ping'
e.salary = 3000

def printInfo(self):
    print(self.name,'\t',self.salary)
e.printInfo = MethodType(printInfo,e)
e.printInfo()
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
Li Ping  3000
```

在类 Employee 的__slots__说明了只允许动态添加三个属性，name、salary、printInfo。例如，动态添加其属性：

```
e.age =23
```

则产生 AttributeError（属性错误）“没有属性'age'”：

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-bda77b7b691a> in <module>()
----> 1 e.age  =23

AttributeError: 'Employee' object has no attribute 'age'
```

抛出的异常错误 AttributeError 说明类 Employee 没有属性 age。也就是说，“e.age =23”试图动态添加一个 age 属性未成功。

2. __slots__的优点

因为不需要使用__dict__管理动态属性，所以可以节省内存、提高速度。下面定义了两个类，类 A 和类 B，其中，类 B 使用了__slots__。


```
class A:
    def __init__(self):
        self.name = 'Li Ping'

class B:
    __slots__ = ['name']
    def __init__(self):
        self.name = 'Li Ping'

import timeit
def f():
    x = A()
    del x

t = min(timeit.repeat(f))
print(t)

def g():
    x = B()
    del x

t = min(timeit.repeat(g))
print(t)
```

输出：

```
0.6455485462242336
0.5542848783678096
```

函数 f()和函数 g()分别创建和删除了类 A 和类 B 的对象，然后用函数 timeit.repeat()来测试调用函数 f()或函数 g()所产生的时间开销。可以看出，使用__slot__的类 B 的对象比类 A 的对象速度要快。

总结

- 可以给一个类或类的对象动态绑定属性。动态绑定方法属性需要使用 types 模块的

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

MethodType()方法。

- 对象的__dict__属性是一个包含了该对象其他所有属性的字典对象。
- __slots__限制一个类能够添加的属性，可节省内存、提高访问速度。

5.5 实战：二叉搜索树

5.5.1 树、二叉树、二叉搜索树

1. 树

树是一个广泛应用的数据结构。例如，日常生活中的组织结构，如政府机构的最高级是中央政府，其下面直接管理着各个省政府，每个省政府下面是各个市政府、每个市政府下面是各个县政府，……同样，一个公司从总部到分部也形成一种树形关系。再如，生物世界中的各种生物物种也形成一种树形关系。如图 5-3 所示是各种生物之间的树形关系。

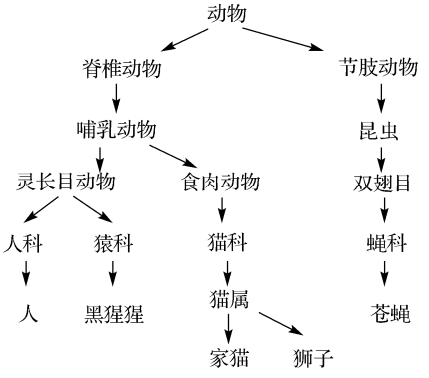


图 5-3 各种生物之间的树形关系

再如，计算机中的文件和文件夹也形成一种树形关系，一个计算机分为多个分区，一个分区下有多个文件(夹)，每个文件夹下又有多个文件(夹)，文件(夹)之间的树形关系如图 5-4 所示。

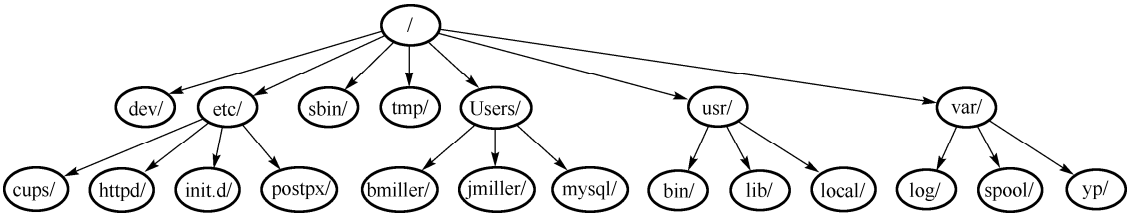


图 5-4 文件(夹)之间的树形关系

同时，一篇文章也可以表示成树形关系，一篇文章可以分为多个段落，一个段落又分为多个句子，一个句子又由一些单词组成。例如，典型的 html 文件也可以表示成不同的 html 元素的一种树形关系：一个 html 文件可以分为 head 和 body 两个元素，而一个 head 又可包含 meta 和 title 元素，同样，一个 body 也由多个元素，如不同的标题 hi、div 区块、p 段落等组成。例如，下面的一个 html 文档：

```
html xmlns="http://www.w3.org/1999/xhtml"
     xml:lang="en" lang="en">
<head>
   <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8" />
   <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
   <li>List item one</li>
   <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

因此，html 文件的元素之间可以表示成一种树形关系(如图 5-5 所示)。

在科学研究中，也经常需要用树来表示某种模型，同时还需要研究不同的针对树模型的算法。例如，微软公司的 kinect 实时体感技术采用了著名的随机森林(多个二叉决策树)技术，学习基于图像的人体骨骼模型，并用于实时检测。再如，在传统的博弈游戏中，如象棋程序，其核心技术就是著名的极小极大(Minimax)树搜索，通过对棋局树的搜索确定最佳的博弈走法。著名的阿尔法狗(AlphaGo)围棋程序使用了改进的蒙特卡洛树搜索(MCTS)表示下棋局面之间的关系，并对棋局和下棋策略进行评估。

一棵树是由一些结点(node)构成的，结点之间具有一种“**一对多**”的“父子关系”，每个结点下面包含多个子结点，它们形成一个父子关系。一个结点既可能是其他结点的孩子，也可能是另外结点的双亲(父亲)。树中只有唯一的根结点，每个结点的子结点个数称为这个结点的“**度**”，树中结点的最大度称为这棵**树的度**。度为“0”的结点称为“**叶子结点**”。如图 5-6 所示是一棵度为“3”的树。

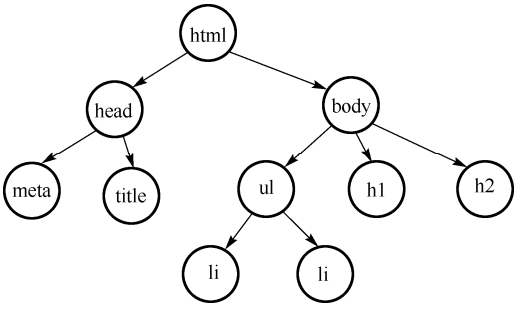


图 5-5　html 文件的元素之间的树形关系

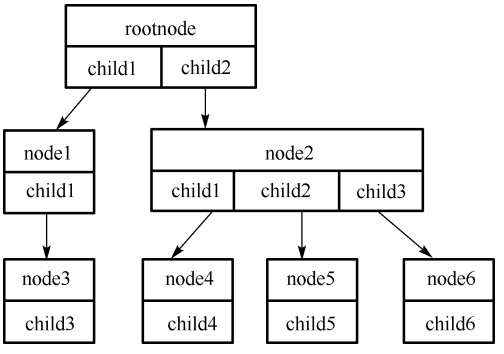


图 5-6　度为“3”的树

结点之间的父子关系称之为“**边**”。

每个结点都可以有一个层次号：一棵树的根结点处于第 1 层，根结点的孩子处于第 2 层，根结点的孩子的孩子处于第 3 层，……一棵树中的结点的最大层次号称为这棵树的深度。一个结点的高度就是这个结点所代表的子树的深度+1。例如，rootnode 代表的树的高度是“3”，node1 代表的树的高度是“2”，而 node3 代表的树只有一个结点，高度为“1”。

2．二叉树

如果树中结点的孩子结点有顺序之分(大儿子、二儿子、三儿子、……)，这样的树称为**有“序树”**。度为“2”的有序树称为“**二叉树**”，即这种树中的每个结点最多只有两个结点，且有顺序之分，结点的两个孩子结点通常称为这个结点的“左孩子”和“右孩子”。二叉树是应用最广泛的一种树，一方面是因为这种树的结构简单易于处理，另一方面是因为一般的树都能转化为二叉树的形式。如图 5-7 所示是一棵二叉树，即每个结点最多有两个孩子(两条边)。

3．二叉搜索树

在一棵二叉树中，如果每个结点表示的数据元素都有一个“键值”，并且对任何一个结点，如果其左子树中的所有结点的键值都小于(或大于)这个结点的键值，其右子树中的所有结点的键值都大于(或小于)这个结点的键值，则这种二叉树就称为“**二叉搜索树**(**Binary Search Tree**)”。如图 5-8 所示是一棵二叉搜索树。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

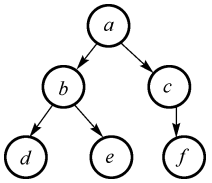


图 5-7　二叉树

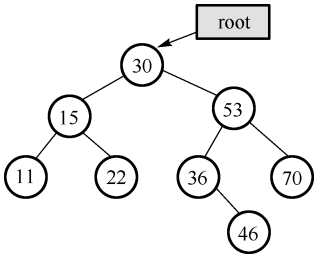


图 5-8　二叉搜索树

如果一棵二叉搜索树的任何结点的左右子树的高度差不超过“1”，那么这棵二叉树就是一棵“**平衡二叉树**”，即任意一个结点的左右子树的高度基本上是一样的。

二叉搜索树应用广泛，如机器学习中的随机森林中的每棵决策树都是一棵二叉搜索树。

树和二叉树是一种递归结构。树和二叉树的定义是一种递归定义，一棵树除树根外，都是树根的子树，而每棵子树都是一棵符合这种定义的树，子树有一个根结点，除这个子树根结点外，都是这个子树根结点的子树。这个定义过程如此递归下去……

5.5.2　树和二叉树的存储表示

在计算机中，表示树或二叉树时既要存储所有的数据元素(结点)，又要表示结点之间的父子关系(边)。

因为树和二叉树都是一种递归的结构，所以表示树和二叉树时采用递归的方法，即对每个结点，既存储这个数据元素的值，又要存储其子结点的存储位置。在 Python 中，可以采用如下两种方法。

1．嵌套的 list

每个数据元素的结点用一个 list 对象表示，其中有该数据元素的值，还有作为其孩子的数据元素结点的 list 对象(每个孩子结点也是一个 list 对象)，即一个结点的 list 中嵌套类其子结点的 list。例如，对于如图 5-7 所示的二叉树，可用下面的 list 对象表示：

```
tree = ['a',                        #根
       ['b',                        #左子树
         ['d', [], []],
         ['e', [], []]
       ],
       ['c',                        #右子树
         ['f', [], []],
         []
       ]
     ]
```

2．结点和引用

表示树的第二种方法是“二叉链表”法或“三叉链表”法，即用一个结点类表示树的一个结点。其中，除它本身的数据元素的值外，还有指向其孩子结点的引用。对于一棵二叉树，首先有一个根结点，根结点中有两个引用指向其左右孩子结点，对于孩子结点也是如此，直到遇到一棵空的子树。这种每个结点包含两个指向左右孩子结点的引用的方法叫作“**二叉链表**”，如图 5-9 所示。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

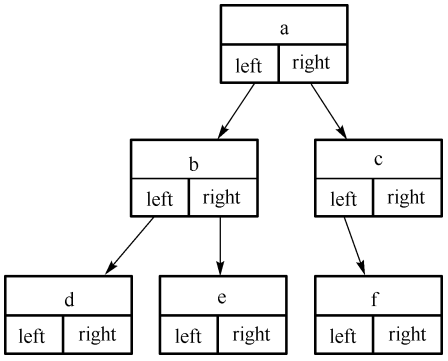


图 5-9　二叉链表

结点用 Python 的类定义可以描述如下：

```
class BiTreeNode:
    def __init__(self,rootObj):
        self.data = rootObj          #数据
        self.leftChild = None        #左孩子引用
        self.rightChild = None       #右孩子引用
```


类 BiTreeNode 表示一个二叉树的结点，其 data 属性存储这个结点本身的数据元素值，其 leftChild 和 rightChild 分别是该结点的左右子树的根结点的引用。可以给这个类添加一些方法，以方便构建结点之间的父子关系。例如：

```
class BiTreeNode:
    def __init__(self,rootObj):
        self.data = rootObj
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self,newNode):
        if self.leftChild == None:   #新结点成为左孩子
            self.leftChild = BiTreeNode(newNode)
        else:      #新结点成为左孩子，原来的左孩子成为新结点的左孩子
            t = BiTreeNode(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BiTreeNode(newNode)
        else:
            t = BiTreeNode(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t

    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setValue(self,obj):
        self.data = obj
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    def getValue(self):
        return self.data
```

可以通过创建这些结点，构造一棵二叉树：

```
#r 指向创建的二叉树的根结点
r = BiTreeNode('a')
r.insertLeft('b')
r.insertRight('c')
b = r.getLeftChild()
b.insertLeft('d')
b.insertRight('e')
r.getRightChild().insertRight('f')
print(r.getLeftChild().getRightChild().getValue())
```

输出：

```
e
```

5.5.3 二叉树的操作

针对一棵二叉树可以有很多操作，如遍历一棵二叉树、查询二叉树的深度、查询二叉树中的叶子结点个数等。其中，最重要的操作就是遍历二叉树，即访问二叉树的每个结点，而且每个结点只访问一次。因为二叉树是一个递归结构，包括一个根、根的左子树、根的右子树，且根的左右子树也是二叉树，所以可以用递归算法写出二叉树的遍历过程。例如：


```
#先根遍历 T 为根的二叉树
def traverse_pre(T):
    if T:
        print(T.data,end = ' ')          #直接访问根
        traverse_pre(T.leftChild)        #遍历 T.leftChild 为根的左子树
        traverse_pre(T.rightChild)       #遍历 T.rightChild 为根的右子树
```

可以测试一下上述函数：

```
traverse_pre(r)
```

输出：

```
a b d e c f
```

上述过程称为“先根遍历”，即在遍历一棵二叉树时总是先访问根，再访问左右子树。还可以使用“后根遍历”，即在遍历一棵二叉树时总是先遍历左右子树，再访问根。代码如下：

```
#后序遍历 T 为根的二叉树
def traverse_post(T):
    if T:
        traverse_post(T.leftChild)       #遍历 T.leftChild 为根的左子树
        traverse_post(T.rightChild)      #遍历 T.rightChild 为根的右子树
        print(T.data,end = ' ')          #直接访问根

#测试这个后根遍历算法
traverse_post(r)
```

输出：

```
d e b f c a
```

当然，还有“中根遍历”，就是先遍历完左子树，然后访问根，最后遍历右子树。代码如下：

```
#中序遍历 T 为根的二叉树
def traverse_mid(T):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        if T:
            traverse_mid(T.leftChild)        #遍历 T.leftChild 为根的左子树
            print(T.data,end = ' ')          #直接访问根
            traverse_mid(T.rightChild)       #遍历 T.rightChild 为根的右子树

    #测试这个中根遍历算法
    traverse_mid(r)
```

输出：

```
    d b e a c f
```

假设中根遍历函数的名字叫作 iot，其遍历过程如图 5-10 所示。

实际上，先根遍历、中根遍历、后根遍历经过的都是同一条路径，遍历过程会三次经过一个结点，如图 5-11 所示。首次遇到该结点就输出其值的是先根遍历；如果从左子树回退到这个结点时才输出该结点的值，就是中根遍历；如果当第三次，即从右子树回到该结点时才输出这个结点，就是后根遍历。

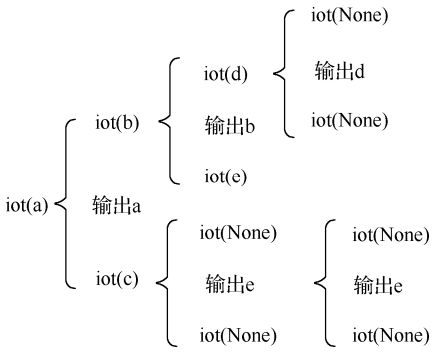


图 5-10　中根遍历函数的过程

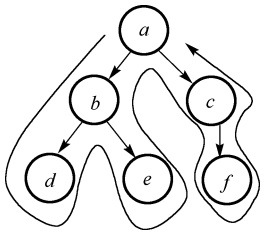


图 5-11　先根遍历、中根遍历、后根遍历是同一条路径

基于遍历算法，可以实现针对二叉树的其他操作。例如，求一棵二叉树的高度(深度)的方法是，先求左子树高度，再求右子树高度，二叉树的最终高度是左右子树高度的最大值+1。代码如下：

```
    #后根遍历求 T 的深度(高度)
    def depth(T):
        if T:
            l = depth(T.leftChild)       #T.leftChild 为根的左子树高度
            r = depth(T.rightChild)      #T..rightChild 为根的右子树高度
            return max(l,r)+1            #左右子树高度的最大值+1
        else:
            return 0

    #测试这个求深度算法
    depth(r)
```

输出：

```
    3
```

5.5.4　二叉搜索树的操作

通常，任何一棵树(包括二叉搜索树)都包括插入、删除、搜索等基本操作，其中，插入、删除可以修改二叉树中的结点及其父子关系，而搜索就是在所有结点中寻找满足某个条件的结点。对于二叉搜索树，由于其结点是按照键值有序的(如左子树结点值<该结点值<右子树结点值)，所以按照

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

键值可以很快地搜索到一个数据元素。

假如有 1024 个数据元素放在一个 list 对象中，若要查找某个键值的元素，则需要从表头到表尾逐个比较，因此需要 1024 次比较才知道是否存在这个数据元素。如果将这些元素存储在一个二叉搜索树中，则只需要从根结点和待查找的键值中进行比较：

- 如果相等，则说明找到；
- 如果“小于”根结点的键值，则在左子树上查找；
- 如果“大于”根结点的键值，则在右子树上查找。

这个过程一直重复，直到找到满足条件的结果，或者到达空树(说明未找到)。这个查找过程所经过的路径就是从根到叶子结点的路径，比较的次数最多不超过树的高度。如果二叉树是平衡二叉树，则其高度不超过“10”，即不超过 10 次就搜索完成了。

上述搜索过程就是二叉树的“先根遍历”过程，因此只要将“先根遍历”代码稍加修改就可以了：

```
def searchBST(T,key):
    if T:
        if key==T.data : return T                          #先搜索根
        elif key<T.data:return searchBST(T.leftChild,key)     #在左子树上搜索
        else: return searchBST(T.rightChild,key)     #在右子树上搜索
    else :return None
```


然而，现在还没有一棵 BST 树，因此基于与查找一样的递归思想，也可以编写在 BST 树上插入一个数据元素的插入函数，通过不断插入数据元素构造一个 BST 树。

在一个结点代表的 BST 树上插入一个数据元素 x 的过程是：如果待插入的值 x 小于结点的值，则在其左子树上插入 x；如果待插入的值 x 大于结点的值，则在其右子树上插入 x；直到遇到一个空树，就确定了插入位置，此时可创建一个新结点，并插入在这个位置。如图 5-12 所示是通过不断插入数据元素构成一个 BST 树的过程。

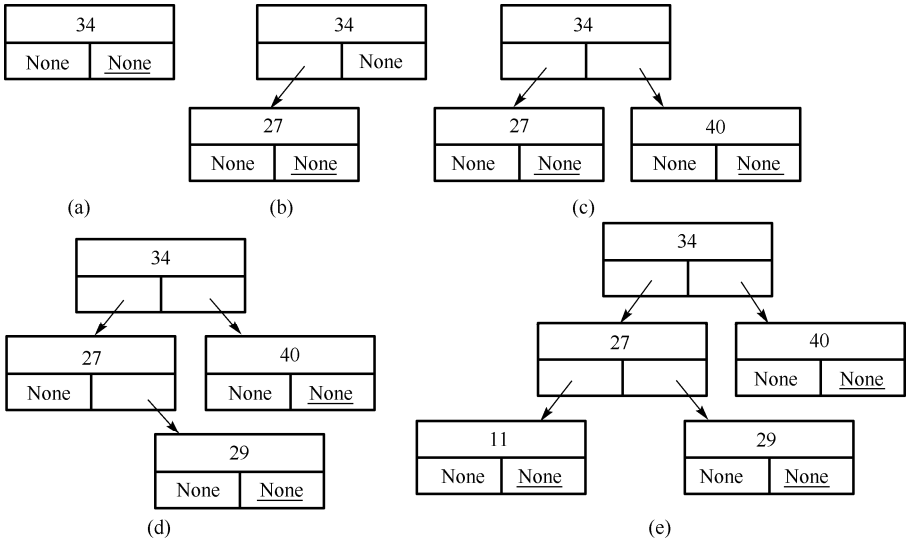


图 5-12　二叉搜索树(BST 树)的插入过程

插入函数的代码如下：

```
#在 T 为根的二叉树插入 x
def insertBST(T,x):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            if T:
                if x<T.data:
                    if T.leftChild:
                        insertBST(T.leftChild,x)      #在 T 的左子树上插入 x
                    else:
                       T.leftChild =  BiTreeNode(x)   #新结点成为 T 的左孩子
                elif x>T.data:
                    if T.rightChild:
                        insertBST(T.rightChild,x)     #在 T 的右子树上插入 x
                    else:
                       T.rightChild =  BiTreeNode(x)  #新结点成为 T 的右孩子

        #不断调用插入函数，构造一个 BST 树
        root = BiTreeNode(34)
        insertBST(root,27)
        insertBST(root,40)
        insertBST(root,29)
        insertBST(root,11)
```

对于一个 BST 树，中根遍历的结果一定是一个有序序列，因此可以用中根遍历新构造的 BST 树，来检查中根序列是否是一个有序序列：


```
        #中根遍历结果应该是一个有序序列
        traverse_mid(root)
        print()
```

输出：

```
        11 27 29 34 40
```

再次测试之前的搜索函数 searchBST()，查看能否正确地搜索：

```
        x = 29
        n = searchBST(root,x)

        if n: print(n.data)
        else: print('未找到:',x)

        x = 15
        n = searchBST(root,x)
        if n: print(n.data)
        else: print('未找到:',x)
```

输出：

```
        29
        未找到: 15
```

总结

- 介绍树、二叉树、二叉搜索树等相关概念，树和二叉树的两种基本存储方式。
- 介绍并实现二叉树的先根遍历、中根遍历、后根遍历，基于遍历操作实现其他的二叉树的操作，如求二叉树的高度。
- 介绍二叉搜索树的插入、查找算法的递归实现。

5.6 实战：面向对象游戏引擎和仿“雷电战机”游戏

前面编写了一个基于函数的过程式游戏引擎框架，并在此基础上编写了一个乒乓游戏。采用面向对象编程技术可以更自然地设计游戏引擎、编写游戏。在一个游戏中，首先有一个代表画布的窗

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

口，窗口本身有各种属性，如大小、标题、背景、颜色等，在窗口里有各种游戏元素，其中的主要元素是各种游戏精灵，即可以自主活动的各种游戏物体，如乒乓游戏里可以运动的球、挡板，再如飞机大战中的飞机、子弹等，这些精灵都是一些具有自主行为能力的对象，它们既可以在相互之间发送或接收消息(射击、碰撞)，也可以接收外界消息(如玩家的鼠标或键盘的输入)。

5.6.1 面向对象游戏引擎

一个游戏初始化后，首先出现的是一个主界面，然后游戏中的精灵们相互作用，同时也接收用户输入，使游戏环境发生变化并用绘制的场景图像显示出来。每个游戏都包含一些共同工作，如初始化、事件处理、更新数据、绘制场景等。每个游戏也具有一些共同数据属性，如游戏画面窗口、游戏中的精灵等对象。可以将所有游戏都共有的这些数据属性和功能属性用一个类表示，这个类可称为“**游戏引擎**”类，因为它控制着整个游戏的运行过程。

这个“游戏引擎”类中包含窗口、精灵等对象，每个对象也都具有自己的数据属性和功能属性，如窗口有长宽、标题、背景和前景颜色等属性，因此可以在窗口的画布上绘制像素等。同时，精灵也有自己的位置、图像、速度、运动等属性。游戏引擎类和窗口、精灵之间是一种包含关系。

1．游戏引擎类

游戏引擎类主要包含的属性有数据属性和功能属性如下。

数据属性：

```
游戏窗口(屏幕)window：窗口长宽、标题、绘制表面、背景或前景图像或颜色、字体颜色等
所有的精灵 Spirit
```

功能属性：

```
init()                    #初始化
    run()                 #游戏主循环
        processEvent()    #事件处理
        collision()       #更新数据并处理碰撞
        render()          #绘制场景
quit()                    #退出游戏
```

因此，可以定义如下的游戏引擎类 GameEngine，其中，构造函数__init__()完成游戏引擎的初始化工作，包括游戏窗口图形环境的初始化和游戏数据的初始化；游戏的主循环 run()方法不断重复四个过程，包括事件处理、更新数据、碰撞检测处理、绘制场景。除这些主要方法外，GameEngine 还定义了一些辅助方法。例如：

```
import random
import pygame, sys
from pygame.locals import *

#----游戏状态常量---------
RUNNING = 1       #是否正在运行
PAUSE = -1        #暂停
STOP = 0          #停止

FPS  = 25  #画面刷新速率

#----颜色 常量---------
BLACK = (0,0,0)
WHITE = (255, 255, 255)
GREEN = (0, 60, 0)
GREY = (210, 210 ,210)
RED = (255, 0, 0)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
PURPLE = (255, 0, 255)
YELLOW = (120,120,0)

class GameEngine():
    #=========1. 初始化=============
    def __init__(self, width, height,title = 'game Engine',background = None,foreground = None):
        self.width, self.height,self.title = width, height,title
        pygame.init()
        pygame.mixer.init()                          #初始化播放声音引擎
        self.surface = pygame.display.set_mode((width, height))
        pygame.display.set_caption(title)            #设置窗口标题
        self.runningState = RUNNING

        #------初始化背景或前景 background and foreground----
        #   self.background = background
        #   self.foreground = foreground
        self.set_background(background)
        self.set_foreground(foreground)
        self.bg_color = BLACK                        #背景颜色
        self.fg_color = YELLOW                       #前景颜色
        self.logo_image = None

        #-------初始化游戏数据---------
        self.sprites_list = []                       #初始化精灵数组
        #...

#=========2. 游戏主循环===========
    def run(self):                                   #游戏介绍画面
        self.game_intro()
        clock = pygame.time.Clock()                  #时钟
        self.running = RUNNING
        while self.running != STOP:
            self.running = self.processEvent()       #2.1 事件处理(键盘、鼠标等)
            if self.running == RUNNING:
                self.update()                        #2.2.1 更新数据
                self.collosion()                     #2.2.2 碰撞检测处理
                self.draw()                          #2.3 绘制场景
                pygame.display.flip()                #pygame.display.update()

        clock.tick(FPS)                              #每秒的帧频率 FPS Frames
        pygame.quit()                                #3. 退出程序

    def game_intro(self):
        pass

    def set_foreground(self, image=None):
        self.foreground = image
        if image:
            self.foreground = pygame.image.load(image)

    def set_background(self, image=None):
        self.background = image
        if image:
            self.background = pygame.image.load(image)

    def init(self):
        pass
    def add_sprite(self,sprite):
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        self.sprites_list.append(sprite)

    def remove_sprite(self,sprite):
        self.sprites_list.remove(sprite)

    #=======2.1 事件处理(键盘、鼠标等)===========
    def processEvent(self):
        for event in pygame.event.get():     #返回当前的所有事件
            if event.type == pygame.QUIT:    #接收窗口关闭事件
                return STOP                  #退出游戏
            elif event.type == KEYDOWN:
                return self.keydown(event)
            elif event.type == KEYUP:
                return self.keyup(event)
        return RUNNING                       #正常

    #  键按下处理函数
    def keydown(self,event):
        if event.key == pygame.K_ESCAPE:
            return STOP     #退出
        return 1            #正常

    # 键弹起处理函数
    def keyup(self,event):
        return 1

    #===========2.2.1 更新数据================
    def update(self):
        for sprite in self.sprites_list:
            sprite.update()
        for sprite in self.sprites_list:
            if sprite.is_dead():
                self.remove_sprite(sprite)

    #===========2.2.2 碰撞检测处理================
    def collosion(self):
        pass

    #===========2.3 绘制场景=========
    def draw(self):
        self.draw_background(self.surface)
        self.draw_foreground(self.surface)
        self.draw_sprites(self.surface)
        self.draw_scores(self.surface)
        #self.player.draw(self.surface)

    def draw_sprites(self, surface):
        for sprite in self.sprites_list:
            sprite.draw(surface)

    def draw_background(self, surface):
        surface.fill(self.bg_color)   # 清空成背景颜色
        if self.background:
            surface.blit(self.background, (0,0))

    def draw_foreground(self, surface):
        if self.foreground:
            surface.blit(self.foreground, (0,0))
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    def draw_scores(self, surface):
        pass

    #在位置 pos 处用字体名 fontname 和字体大小 fontsize 绘制文本 txt
    def draw_text(self,text,pos,fontname = "Comic Sans MS", fontsize=20):
        #绘制得分 scores
        myfont1 = pygame.font.SysFont(fontname, fontsize)
        label1 = myfont1.render(text, 1, (255,255,0))
        self.surface.blit(label1, pos)
```

更新数据函数 update()和碰撞处理函数 collission()分开处理。碰撞处理的结果会导致游戏中数据(精灵状态，如位置或速度等)的修改。draw_sprites()是绘制所有精灵图像的函数，它调用了每个精灵自身的绘制函数，并将绘制表面(屏幕/画布)传递给该函数。

创建一个 GameEngine 对象，然后调用其 run()方法后就开始运行游戏：

```
#--------主程序--------
WIDTH = 640
HEIGHT = 480
game = GameEngine(WIDTH,HEIGHT)
game.run()
```


2. 精灵

精灵是游戏中的主要角色，有各种各样的精灵(如挡板、球、飞机、炮弹等)，可以定义一个最基本的精灵类，以表示所有不同种类精灵的共同特性，并在此基础上再派生出特殊的精灵类。

精灵的共同属性包括(图片表示的)形象、位置、绘制(在屏幕上显示自己)、更新状态。

下列代码定义一个精灵类 Sprite 而没有用 Python 自带的精灵类 Sprite，这是为了使读者能够从底层了解游戏引擎的原理。

```
#=============所有精灵类的基类=================
class Sprite():
    def __init__(self, image=None, pos=(0,0),vel = (0,0)):
        self.pos = [pos[0],pos[1]]
        self.vel = [vel[0],vel[1]]
        self.lives = 1
        self.set_image(image)

    #----更新状态---
    def update(self):
        # 更新位置
        self.pos[0] += self.vel[0]
        self.pos[1] += self.vel[1]

        self.rect.x = int(self.pos[0]-self.rect.width//2)
        self.rect.y = int(self.pos[1]-self.rect.height//2)

    def draw(self, surface):
        if self.image:
            surface.blit(self.image, self.rect)
        else:
            pygame.draw.rect(surface, WHITE, self.rect)

    def  colliderect(self,sprite):
        return self.rect.colliderect(sprite.rect)

    def is_dead(self):
        return self.lives == 0
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    def set_image(self,image):
        self.image = image
        if  image:
            self.image = pygame.image.load(image)
            image_rect = self.image.get_rect()
            self.rect = pygame.rect.Rect(self.pos[0]-image_rect.width//2,
                            self.pos[1]-image_rect.height//2,
                            image_rect.width,image_rect.height)
```

类 Sprite 初始化自己的位置(pos)、速度(vel)、图片(image)、生命值(lives)、包围矩形(rect)，update()方法和draw()方法分别更新自身的位置和在屏幕对象(surface)上绘制自己，函数colliderect()用于检测是否和其他的精灵发生了碰撞。

5.6.2 Pong 游戏

本节说明如何用面向对象编程方法在上述的面向对象游戏引擎框架下开发一个具体的游戏。一个具体的游戏通常有多种不同的精灵对象，以前面讲述的 Pong 游戏为例，Pong 游戏中除代表整个游戏的引擎对象外，另外还有两种精灵，即表示乒乓球和挡板的精灵。

在上述类 Sprite 的基础上，可以自定义类 Ball 表示乒乓球，类 Paddle 表示挡板。


```
class Ball(Sprite):
    def __init__(self, radius=15,pos=(0,0),image=None):
        super().__init__(image, pos)     #调用父类的构造函数
        self.init_velocity()
        self.radius = radius
        if not image:
            self.rect = pygame.rect.Rect(pos[0]-radius, pos[1]-radius,
                2*radius, 2*radius)

    def init_velocity(self):
        horizontal = random.randrange(12, 24)/30
        vertical = random.randrange(6, 18)/30
        if random.random()>0.5:          #改变水平速度方向
            horizontal= -horizontal
        if random.random()>0.5:          #改变垂直速度方向
            vertical= -vertical
        self.vel = [horizontal,-vertical]

    def draw(self, surface):
        if self.image:
            surface.blit(self.image, self.rect)
        else:
            pygame.draw.circle(surface, WHITE, (int(self.pos[0]),
                                int(self.pos[1])), self.radius, 0)

     #----更新状态---
    def update(self):
        super(Ball,self).update()


    def backward(self,horizonal = 1):
        if horizonal==1:
            self.vel[0] = - self.vel[0]* 1.1
        elif horizonal==-1:
            self.vel[1] = - self.vel[1]* 1.1
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            else :
                self.vel[0] = -self.vel[0]* 1.1
                self.vel[1] = -self.vel[1]* 1.1
```

在类 Ball 中的 update()方法中调用了父类的 update()方法用于更新自己的位置。如果乒乓球和墙(或档板)发生碰撞就调用反弹函数 backward()，反弹函数 backward()根据反弹的方向修改水平、垂直或两个方向的速度。函数 init_velocity()用来生成一个随机的速度。而如果没有图像的话，函数 draw()就是画一个圆表示自己。例如：

```
class Paddle(Sprite):
    def __init__(self,size=(8,80),pos=(0,0),image=None):
        super().__init__(image, pos)
        self.size = [size[0],size[1]]
        if not image:
            self.rect = pygame.rect.Rect(pos[0]-size[0]//2, pos[1]-size[1]//2,
                                         size[0], size[1])

    def draw(self, surface):
        if self.image:
            surface.blit(self.image, self.rect)
        else:
            GREEN = (0,255,0)
            pygame.draw.rect(surface, GREEN, (self.rect.x, self.rect.y,
                                         self.rect.width, self.rect.height))
```


类 Paddle 并没有定义自己的 update()方法，说明直接继承了父类 Sprite 的 update()方法。而如果没有图像的话，函数 draw()就是画一个矩形表示自己。

除两个精灵类 Ball 和 Paddle 外，还需要在通用的游戏引擎类 GameEngine 基础上，派生定义一个表示 Pong 游戏的游戏引擎类，假设叫作 PongGame。例如：

```
class PongGame(GameEngine):
    def __init__(self, width, height,paddle_width=8,paddle_height=80,
                       ball_radius = 15,title = 'Pong Game',\
                  background = None,foreground = None):
        super().__init__( width, height,title,background,foreground)

        self.scores = [0,0]
        self.ball = Ball(ball_radius,[width//2, height//2])

        self.pad_width = paddle_width
        self.pad_height = paddle_height
        self.half_pad_width = paddle_width//2
        self.half_pad_height = paddle_height//2

        self.paddle1 = Paddle((self.pad_width,self.pad_height),
                       (self.half_pad_width,height//2))
        self.paddle2 = Paddle((self.pad_width,self.pad_height),
                       (width-self.half_pad_width,height//2))
        self.sprites_list = [self.ball,self.paddle1,self.paddle2]

        self.paddle_pressed = 0

    def collosion(self):
        ball = self.sprites_list[0]
        paddle1 = self.sprites_list[1]
        paddle2 = self.sprites_list[2]
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        width,height = self.surface.get_size()

        #与上下墙碰撞，水平速度不变，垂直速度相反
        if ball.pos[1] < ball.radius or ball.pos[1] > height - 1 - ball.radius:
            ball.backward(-1)

        if ball.pos[0] < (ball.radius + paddle1.rect.width):
            if ball.pos[1] <= paddle1.pos[1] + self.half_pad_height and \
               ball.pos[1] >= paddle1.pos[1] - self.half_pad_height:
                    ball.backward(1)
                    self.scores[0] += 1
            else:
                ball.pos = [self.width//2,self.height//2]
                ball.init_velocity()
                self.scores[1] += 1

        if ball.pos[0] >  self.width - 1 - ball.radius - self.pad_width:
            if ball.pos[1] <= paddle2.pos[1] + self.half_pad_height and  \
               ball.pos[1] >= paddle2.pos[1] - self.half_pad_height:
                    ball.backward(1)
                    self.scores[1] += 1
            else:
                ball.pos = [self.width//2,self.height//2]
                ball.init_velocity()
                self.scores[0] += 1

        # 更新挡板位置，保持挡板在窗口里
        if paddle1.pos[1] < self.half_pad_height or paddle1.pos[1]
                  +self.half_pad_height> self.height:
            paddle1.pos[1] -=  paddle1.vel[1]

        if paddle2.pos[1] < self.half_pad_height or paddle2.pos[1]
                  +self.half_pad_height> self.height:
            paddle2.pos[1] -=  paddle2.vel[1]

    # 按键按下的处理函数
    def keydown(self,event):
        if event.key == pygame.K_ESCAPE:  #退出
            return STOP
        elif event.key == pygame.K_SPACE:  #暂停
            return PAUSE
        elif event.key == K_UP:
            self.paddle2.vel[1] = -PAD_MOVE_OFFSET
            self.paddle_pressed = 2
        elif event.key == K_DOWN:
            self.paddle2.vel[1] = PAD_MOVE_OFFSET
            self.paddle_pressed = 2
        elif event.key == K_w:
            self.paddle1.vel[1] = -PAD_MOVE_OFFSET
            self.paddle_pressed =1
        elif event.key == K_s:
            self.paddle1.vel[1] = PAD_MOVE_OFFSET
            self.paddle_pressed = 1
        return RUNNING
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        # 按键弹起的处理函数
        def keyup(self,event):
            if event.key in (K_w, K_s):
                self.paddle1.vel[1] = 0
            elif event.key in (K_UP, K_DOWN):
                self.paddle2.vel[1] = 0
            return RUNNING

        #绘制得分
        def draw_scores(self, surface):
            self.draw_text(str(self.scores[0]),(10,10))
            self.draw_text(str(self.scores[1]),(self.width-30,10))
```

输入参数 paddle-width，paddle-height 分别表示挡板的宽和高，ball-radius 表示球的半径。根据这些参数，构造函数构造了表示左右挡板类 Paddle 的对象 paddle1、paddle2 和表示球类 Ball 的对象 ball，并将它们添加到精灵列表对象 spriteslist 中。scores 表示左右挡板的得分。collosion()方法处理球和墙壁、挡板的碰撞反弹，并限制挡板不要移出上下墙外。函数 keydown()和函数 keyup()分别处理按键按下和按键弹起的情况，如移动挡板或退出程序。

最后，可以用一个主程序，测试这个新的游戏 PongGame。例如：


```
#--------主程序--------
WIDTH = 640
HEIGHT = 480
win = PongGame(WIDTH,HEIGHT)
win.run()
```

如图 5-13 所示是 Pong 游戏的运行截图。

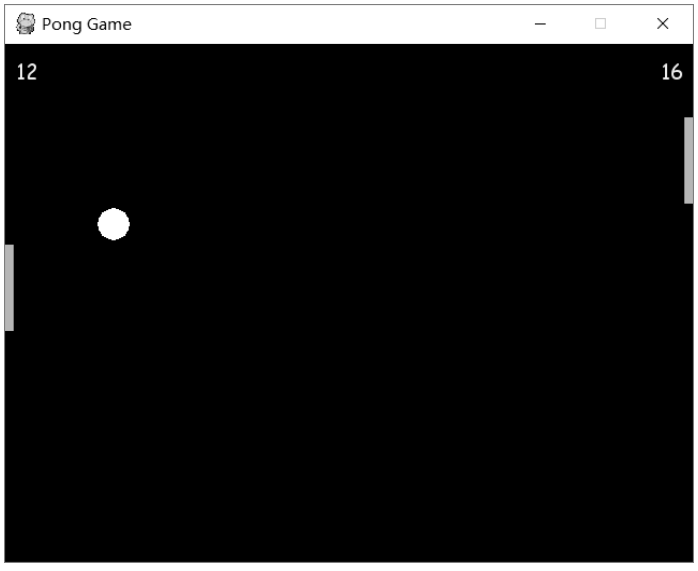


图 5-13　Pong 游戏的运行截图

5.6.3　仿“雷电战机”游戏

在仿“雷电战机”游戏中，敌我双方战机、爆炸、声音等资源文件很多，为此，专门定义了一个类 GameResource 用于保存图像、声音文件的路径。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
class GameResource:
    def __init__(self,init_file = None):
        img_dir = 'SpaceInvander/images/'
        sound_dir = 'SpaceInvander/sounds/'
        self.bg_fg = ['map.jpg','fg_map.jpg']
        self.player_images = ['player1.png','player2.png','player1_hit1.png',
           'player1_hit2.png','player2_hit1.png','player2_hit2.png',
           'player_destroy1.png','player_destroy2.png','player_destroy3.png']
        self.enemy_images = ['Enemy1.png','Enemy2.png','Enemy3.png',
           'Enemy1_hit1.png','Enemy1_hit2.png','Enemy2_hit1.png','Enemy2_hit2.png',
           'Enemy3_hit1.png','Enemy3_hit2.png','Enemy1_destroy1.png','Enemy1_destroy2.png',
           'Enemy2_destroy1.png','Enemy2_destroy2.png','Enemy3_destroy1.png',
           'Enemy3_destroy2.png']
        self.boss_images = ['Boss1.png','Boss2.png','Boss1_hit1.png',
                        'Boss1_hit2.png','Boss2_hit1.png','Boss2_hit2.png',
                        'Boss_destroy1.png','Boss_destroy2.png',
                         'Boss_destroy3.png','Boss_destroy4.png',
                         'Boss_destroy5.png','Boss_destroy6.png',
                         'Boss_destroy7.png','Boss_destroy8.png',
                         'Boss_destroy9.png','Boss_destroy10.png']

        self.bullet_images = ['bullet_blue.png','bullet_purple.png',
                        'bullet_yellow.png']
        self.enemy_bullet_images = ['EnemyBullet1.png','EnemyBullet2.png',
                        'EnemyBullet3.png']


        self.bg_fg = [img_dir+x for x in self.bg_fg]
        self.player_images = [img_dir+'player_images/'+x for x in self.player_images]
        self.enemy_images = [img_dir+'enemy_images/'+x for x in self.enemy_images]
        self.boss_images = [img_dir+'enemy_images/'+x for x in self.boss_images]
        self.bullet_images = [img_dir+'bullet_images/'+x for x in self.bullet_images]
        self.enemy_bullet_images = [img_dir+'bullet_images/'+x for x in
                                self.enemy_bullet_images]

        self.sounds = {'bg_music':'bg_music.mp3','shot_sound':'shot.wav',
                     'player_hited_sound':'user_down.wav',
                     'crash_sound':'explosion.wav',
                     'explosion_sound':'explosion.wav','user_down_wav':'user_down.wav'
                    }

        self.sounds.update((k, sound_dir+v ) for k,v in self.sounds.items())
```

游戏引擎类 SpaceInvander 从类 GameResource 的对象接收这些资源文件的路径，并初始化相应的数据。creat_sprites()方法用于创建游戏中的精灵，draw_background()方法用于绘制游戏的背景画面。同时，为了让背景不断移动，在原始背景图片的上方再拼接一个同样的图片，当原始背景图片向下移动时，上方的图片也一起向下移动，一旦原始图片和复制图片整个移出或移入，再让它们回到起始位置，重复这种移动，造成一个无限大的背景的错觉。初始化程序在读取了背景音乐文件后就开始播放背景音乐。例如：

```
from pygame.time import get_ticks
from pygame import mixer
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

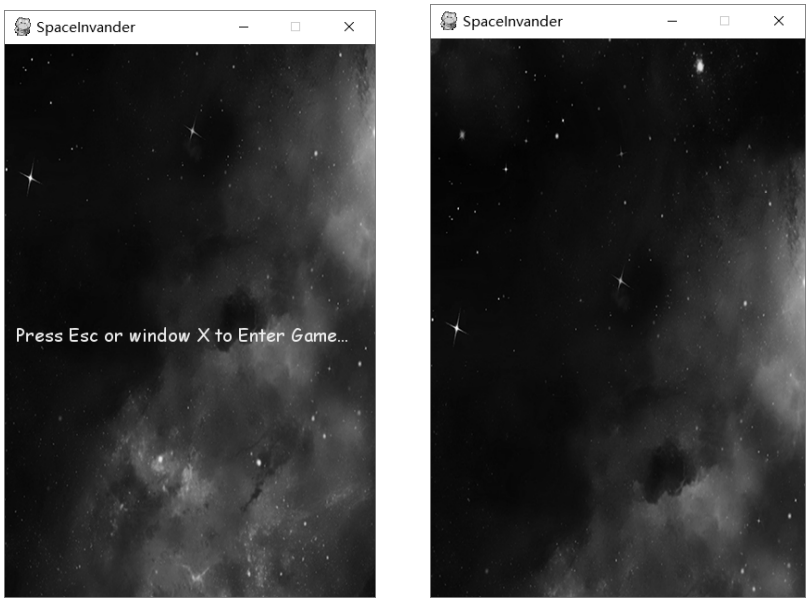
```
class SpaceInvander(GameEngine):
    def __init__(self, width, height,title='SpaceInvander',resource=GameResource()):
        super().__init__( width, height,title)
        self.resource =resource
        self.background = pygame.image.load(self.resource.bg_fg[0]).convert()
        self.background2 = self.background.copy()
        self.pos_y1  = -1024
        self.pos_y2  = 0
        self.bg_roll_speed_factor = 0.1

        #播放背景音乐
        mixer.init()   #sound
        pygame.mixer.music.load(self.resource.sounds['bg_music'])
        pygame.mixer.music.play(1)

        self.player_id = 0
        self.player = None
        self.enemy_id = 0
        self.player = None
        self.boss_id = 0
        self.boss = None
        self.aliens = []
        self.bullets = []
        self.bullets_boss = []
        self.explosions = []
        self.score = 0
        self.creat_sprites()

    def creat_sprites(self):
        pass

    def draw_background(self, surface):
        surface.fill(self.bg_color)          #clear surface to green
        surface.blit(self.background, (0,0))
        surface.blit(self.background, (0, self.pos_y1))
        surface.blit(self.background2, (0, self.pos_y2))

    def update(self):
        super().update()
        self.pos_y1 += self.bg_roll_speed_factor
        self.pos_y2 += self.bg_roll_speed_factor

        if self.pos_y1 > 0:
            self.pos_y1 = -1024
        if self.pos_y2 > 1024:
            self.pos_y2 = 0

    def game_intro(self):
        logo_img = self.background
        intro = True
        while intro:
            #检查是否退出
            for event in pygame.event.get():
                if event.type == pygame.QUIT or event.type ==
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                    pygame.KEYDOWN and event.key==pygame.K_ESCAPE:
                intro = False
                break

            self.surface.blit(logo_img, (0, 0))
            text = "Press Esc or window X to Enter Game..."
            self.draw_text(text,(12,self.height//2))
            pygame.display.flip()           #pygame.display.update()
        pygame.mixer.music.stop()

    def draw_scores(self, surface):
        self.draw_text(str(self.score),(self.width-30,10))

WIDTH = 400
HEIGHT = 600
resource = GameResource()
game = SpaceInvander(WIDTH,HEIGHT,'SpaceInvander',resource)
game.run()
```

运行上述程序，run()方法将首先执行 game_intro()方法，出现如图 5-14(a)所示的启动画面，并播放背景音乐。按下 Esc 键或用鼠标点击窗口右上角的×，即可停止音乐播放，并进入背景变化的游戏开始画面(如图 5-14(b)所示)。

游戏的精灵主要包括玩家(我方)战机 Player、敌方母机 Boss、敌方战机、敌方子弹、我方子弹。除具有精灵已有的属性外如位置(pos)、速度(vel)、更新(update())、绘制(draw())，我方战机、敌方战机都具有一些共同属性，包括生命值(lives)、射击(shot())、死亡判断函数(is_dead())。因此，需要首先定义一个从类 Sprite 派生的表示战机的精灵类 Fighter。例如：



(a) 启动画面　　(b)游戏开始画面

图 5-14　仿“雷电战机”开始画面

```
class Fighter(Sprite):
    def __init__(self,image=None, hit_images=None,destroy_images=None,
                 bullet_images=None,shot_sound = None,crash_sound = None,
                 bullet_speed = 1,move_speed = 0.5,lives=3,power = 3,pos=(0,0)):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            super().__init__(image, pos)
            self.hit_images = hit_images              #射中时的图像
            self.destroy_images = destroy_images      #销毁图像
            self.lives = lives                        #生命值
            self.power = 1                            #战斗力
            self.move_speed  = move_speed             #运动速度系数

            self.bullet_images = bullet_images
            self.bullet_id = 0                        #战斗力增强后可以升级子弹
            self.bullet = None
            self.bullet_speed = bullet_speed
            self.shot_sound = shot_sound

            self.crash_sound = crash_sound
            self.hit_id=0

        def shot(self):
            self.bullet = Bullet(self.bullet_images[self.bullet_id],
                      (self.rect.centerx, self.rect.top),(0,self.bullet_speed) )
            if self.shot_sound:
                self.shot_sound.play()
            return self.bullet

        def is_dead(self):
            return self.lives==0

        def hitted(self):        #被击中时的处理
            self.lives -= 1
            if self.crash_sound:
                self.crash_sound.play()
            if self.hit_id<len(self.hit_images):
                self.set_image(self.hit_images[self.hit_id])
            self.hit_id +=1
```


同样，可以为敌我双方的子弹定义一个共同的子弹类 Bullet。例如：

```
class Bullet(Sprite):
    def __init__(self,image=None, pos=(0,0),velocity = (0,0)):
        super().__init__(image, pos)
        self.pos[1] -= self.rect.height//2
        self.rect.y -= self.rect.height//2
        self.vel = [velocity[0],velocity[1]]

    def update(self):
        super().update()
        w, h = pygame.display.get_surface().get_size()
        if self.rect.y < 1 or self.rect.y>= h:
            self.lives = 0
```

玩家 Player 除具有 Fighter 的共同属性外，还具有加油（powUp()）功能。同时，可以修改（update()）位置更新方法，以防止 Player 跑出窗口区域。例如：

```
class Player(Fighter):
    def powerUp(self):
        self.power += 1
        self.power_time = pygame.time.get_ticks()

    def update(self):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            # 更新位置
            w, h = pygame.display.get_surface().get_size()
            x = self.pos[0] + self.vel[0]
            y = self.pos[1] + self.vel[1]
            if x>self.rect.width//2 and x< w- self.rect.width//2:
                self.pos[0] = x
            if y>self.rect.height//2 and y< h- 20+1:
                self.pos[1] = y
            self.rect.x = int(self.pos[0]-self.rect.width//2)
            self.rect.y = int(self.pos[1]-self.rect.height//2)
```

现在修改类 SpaceInvander（或定义一个派生类 SpaceInvander2），在 creat_sprites() 方法里添加一个类 player，然后定义事件处理函数，当按空格键时就发射子弹，按移动键时就移动。例如：

```
class SpaceInvander2(SpaceInvander):
    #...
    def creat_player(self):
            image = self.resource.player_images[self.player_id]
            self.num =2
            hit_id =  self.num+2*self.player_id # 2*(self.player_id+1)
            hit_images = [self.resource.player_images[hit_id],
                                self.resource.player_images[hit_id+1]]
            des_id = self.num+2*self.num+2*self.player_id
            des_images = [self.resource.player_images[des_id],
                                self.resource.player_images[des_id+1]]

            shot_sound = mixer.Sound(self.resource.sounds['shot_sound'])
            crash_sound = mixer.Sound(self.resource.sounds['crash_sound'])

            shoot_delay = 250
            bullet_speed = -1
            move_speed= 0.5
            lives=3
            power = 3
            pos= (self.width//2, self.height-20)

            self.player = Player(image,hit_images, des_images,
                                 self.resource.bullet_images,shot_sound,
                                 crash_sound,bullet_speed, move_speed,lives,power,pos)
    def creat_sprites(self):
        self.creat_player()
        self.add_sprite(self.player)

     # keydown handler
    def keydown(self,event):
        if event.key == pygame.K_ESCAPE:  #退出
            return 0
        elif event.key == pygame.K_SPACE:
            bullet = self.player.shot()
            if bullet:
                self.bullets.append(bullet)
                self.add_sprite(bullet)
                self.player.bullet = None

        elif event.key == K_RIGHT:
            self.player.vel[0] = self.player.move_speed
            #self.player.update()
        elif event.key == K_LEFT:
            self.player.vel[0] = -self.player.move_speed
        elif event.key == K_DOWN:
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                self.player.vel[1] = self.player.move_speed
            elif event.key == K_UP:
                self.player.vel[1] = -self.player.move_speed
            elif event.key == K_w:
                pass
            elif event.key == K_s:
                pass
            return 1

        #keyup handler
        def keyup(self,event):
            if event.key in (K_w, K_s):
                pass
            elif event.key in (K_RIGHT, K_LEFT):
                self.player.vel[0] = 0
            elif event.key in (K_DOWN,K_UP):
                self.player.vel[1] = 0
            return 1

        def update(self):
            super().update()

            if self.player and self.player.is_dead():
                    self.player = None

            for bullet in self.bullets:
                if bullet.is_dead():
                    self.bullets.remove(bullet)

    game = SpaceInvander2(WIDTH,HEIGHT)
    game.run()
```

图 5-15　游戏画面

运行这个程序后将显示如图 5-15 所示的游戏画面。此时，我方的战机可以发射子弹，但还没有敌机，得分为“0”。

假设敌方战机有两种，普通的敌方战机和敌方大 Boss 战机。游戏开始时，随机出现的是一些敌方战机，每个敌方战机会发射子弹，当我方战机击溃一定数目的敌方战机后，敌方大 Boss 战机就会出现，敌方大 Boss 战机同样会发射子弹，并缓慢地向我方逼近。当然，敌方大 Boss 战机发射的子弹类型可能和我方的子弹类型不必完全相同，但本书为了简单起见，重复利用同一个类 Bullet 表示双方的子弹，只是给它们不同的形状。

因此，从战机类 Fighter 派生出类 Enemy 和类 Boss 分别用于表示敌方普通战机和敌方大 Boss 战机。敌机移动和发射子弹不是由用户控制的，而是其自主的行为，因此，在 update() 方法里让敌方战机随机移动和发射子弹，即让敌方战机随机移动，经过一定时间后开始射击。例如：

```
class Enemy(Fighter):
    def __init__(self,image=None, hit_images=None,destroy_images=None,
                 bullet_images=None,shot_sound = None,crash_sound = None,
                 shoot_delay= 250,shoot_interval = 2500,bullet_speed = 1,move_speed = 0.5,
                 lives=3,power = 3,pos=(0,0)):
        super().__init__(image,hit_images,destroy_images,bullet_images,shot_sound,
                         crash_sound,bullet_speed ,move_speed,lives,power,pos)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            self.shoting = False                        #是否射击标志
            self.shoot_delay = shoot_delay              #连续射击延迟
            self.shoot_interval = shoot_interval        #两次射击的间隔
            self.shot_last = pygame.time.get_ticks()
            self.shot_start = pygame.time.get_ticks()

        def update(self):
            self.shot()
            self.generate_velocity()
            super().update()

        def generate_velocity(self):
            move = random.random()>0.8
            acc = random.random()>0.5
            if(move):
                self.vel[1] = random.random()*self.move_speed
                self.vel[0] = (random.random()-0.5)*self.move_speed
                if acc: self.vel[0] *=5

        def shot(self):
            now = pygame.time.get_ticks()
            if now - self.shot_start > self.shoot_interval:    #超过一定时间，改变射击标志
                self.shoting = not self.shoting
                self.shot_start = now
            if not self.shoting: return None
            if now - self.shot_last > self.shoot_delay: #开始射击，修改最后一次射击时间
                super().shot()
                self.shot_last = now

    class Boss(Enemy):
        pass
```

现在，需要修改 SpaceInvander 类(或定义一个派生类 SpaceInvander3)的 creat_sprites()方法、creat_enemies()方法和 creat_Boss()方法，以及 update()方法。

```
    import copy

    def copy_enemy(enemy,enemy2):
            enemy2.vel = copy.copy(enemy.vel)
            enemy2.pos = copy.copy(enemy.pos)
            enemy2.rect = copy.copy(enemy.rect)


    class SpaceInvander3(SpaceInvander2):
        def creat_sprites(self):
            super().creat_sprites()
            self.boss = None
            self.enemys = []
            self.bullets_enemy = []
            self.enemy_last_time =  pygame.time.get_ticks()
            self.enemy_time_interval = 5000

        def creat_enemies(self):
            if self.running != RUNNING: return
            self.enemy_id = 0
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            self.num = 3
            image = self.resource.enemy_images[self.enemy_id]
            hit_id = self.num+2*self.enemy_id
            hit_images = [self.resource.enemy_images[hit_id],
                          self.resource.enemy_images[hit_id+1]]
            des_id = self.num+2*self.num+ 2*self.enemy_id
            des_images = [self.resource.enemy_images[des_id],
                          self.resource.enemy_images[des_id+1]]
            shot_sound = None #mixer.Sound(self.resource.sounds['shot_sound'])
            crash_sound = mixer.Sound(self.resource.sounds['crash_sound'])
            shoot_delay= 500
            shoot_interval = 3000

            bullet_speed = 0.5
            move_speed= 0.2
            lives=1
            power = 1
            pos= (self.width/2, 20)

            enemy = Enemy(image,hit_images, des_images,self.resource.enemy_bullet_images,
                        shot_sound,crash_sound,shoot_delay,shoot_interval,bullet_speed,
                        move_speed,lives,power,pos)
            enemy.image = pygame.transform.rotate(enemy.image, 180)
            enemy.last_shot_time = pygame.time.get_ticks()
            enemy.shot_delay = 1000

            enemy.pos[0] = self.width/2+ (random.random()-0.5)*0.2*self.width     #x 随机偏移
            enemy.pos[1] = random.random()*self.num *enemy.rect.height            #y 随机偏移
            # self.add_sprite(enemy)
            # return enemy

            enemy2 = copy.copy(enemy)
            enemy3 = copy.copy(enemy)
            copy_enemy(enemy,enemy2)
            copy_enemy(enemy,enemy3)

            enemy2.pos[0] = random.uniform(enemy.rect.width*2,
                          enemy.pos[0]-enemy.rect.width)
            enemy2.pos[1] += (random.random()-0.5)*enemy.rect.height*0.5
            enemy3.pos[0] = random.uniform( enemy.pos[0]+enemy.rect.width,
                          self.width-enemy.rect.width*2)
            enemy3.pos[1] += (random.random()-0.5)*enemy.rect.height*0.5

            enemys = [enemy,enemy2,enemy3]
            for e in enemys:
                e.update()
                self.add_sprite(e)
            self.enemys += enemys



        def creat_Boss(self):
            if self.running != RUNNING: return
            self.boss_id = 0
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            image = self.resource.boss_images[self.boss_id]
            hit_id = self.num+2*self.boss_id
            hit_images = [self.resource.boss_images[hit_id],
                          self.resource.boss_images[hit_id+1]]
            des_id = self.num+2*self.num+ 2*self.boss_id
            des_images = [self.resource.boss_images[des_id],
                          self.resource.boss_images[des_id+1]]
            shot_sound = None #mixer.Sound(self.resource.sounds['shot_sound'])
            crash_sound = mixer.Sound(self.resource.sounds['crash_sound'])
            shoot_delay= 250
            shoot_interval = 2500
            bullet_speed = 1
            move_speed= 0.1
            lives=1
            power = 1
            pos= (self.width/2, -30)
            self.boss = Boss(image,hit_images, des_images,self.resource.enemy_bullet_images,
                  shot_sound,crash_sound,shoot_delay,shoot_interval,bullet_speed,
                  move_speed,lives,power,pos)

            self.add_sprite(self.boss)

        def update(self):

            for enemy in self.enemys:
                if enemy.pos[1]>self.height:
                    enemy.lives=0
                    continue
                if enemy.bullet:
                    self.add_sprite(enemy.bullet)
                    self.bullets_enemy.append(enemy.bullet)
                    enemy.bullet = None

            if self.boss:
                if self.boss.pos[1]>self.height:
                    self.remove_sprite(self.boss)
                    self.boss = None
                elif self.boss.bullet:
                    self.add_sprite(self.boss.bullet)
                    self.bullets_enemy.append(self.boss.bullet)
                    self.boss.bullet = None
            elif(self.score>5):
                self.creat_Boss()   #创建敌方大 Boss 战机

            for enemy in self.enemys:
                if enemy.is_dead():
                    self.enemys.remove(enemy)
                    #self.remove_sprite(enemy)

            for bullet in self.bullets_enemy:
                if bullet.is_dead():
                    self.bullets_enemy.remove(bullet)
                    #self.remove_sprite(bullet)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            for bullet in self.bullets:
                if bullet.is_dead():
                    self.bullets.remove(bullet)
                    #self.remove_sprite(bullet)

            #新创建一批敌机
            now =  pygame.time.get_ticks()
            if now - self.enemy_last_time> self.enemy_time_interval:
                self.creat_enemies()   #经过一段时间，创建一批敌机
                self.enemy_last_time = now

            super().update()

    game = SpaceInvander3(WIDTH,HEIGHT)
    game.run()
```

运行上述程序，此时敌我双方战机都可发射子弹，并将出现如图 5-16 所示的画面。

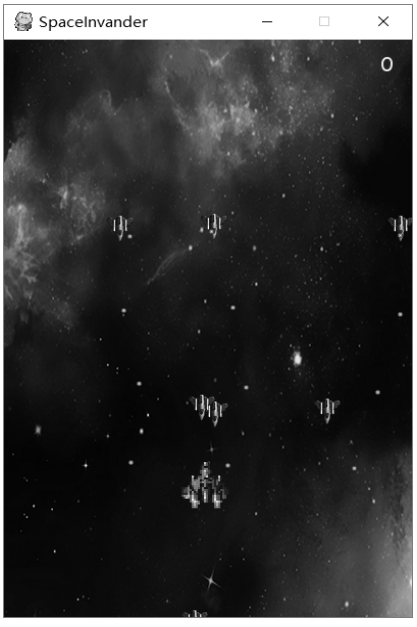


图 5-16　敌我双方战机都可发射子弹

但是，此时的程序当子弹射向敌方战机、敌方战机射向我方战机或发送碰撞时，都没有做任何处理。以下增加碰撞处理，当子弹击中目标时，产生一个爆炸效果，并销毁被击中目标。首先需要定义一个爆炸精灵，爆炸精灵 Explosion 模拟爆炸的动画效果，因此需要多幅图片，当然可以用 Pygame 的精灵类轻松完成这个爆炸动画效果。这里从头编写用多个图片模拟动画效果，以便读者能够更好地从底层理解实现原理。Explosion 构造函数接收一个多个图片路径的 list 对象：

```
    class Explosion(Sprite):
        def __init__(self, images, position,explosion_wav):
            self.images = images
            super(Explosion, self).__init__(images[0], position)
            self.frame = 0
            self.last_update = pygame.time.get_ticks()
            self.frame_rate = 50
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            self.lives = len(self.images)
            self.sound_explosion = mixer.Sound(explosion_wav)

        def update(self):
            now = pygame.time.get_ticks()
            if now - self.last_update > self.frame_rate: #经过一定时间，更新爆炸画面
                self.last_update = now
                self.frame += 1
                if self.frame == len(self.images):   #爆炸动画图片播放完，炸弹就销毁
                    self.lives = 0
                else:
                    self.set_image( self.images[self.frame])
                    super().update()
```

以下添加碰撞检测功能，当发生碰撞时，生命值减少“1”，当生命值为“0”时，就销毁并产生爆炸效果：

```
    class SpaceInvander4(SpaceInvander3):

        def collosion(self):
            for enemy in self.enemys:  #对每个敌机 enemy
                player = self.player
                if player and player.colliderect(enemy): #是否和我方战机碰撞
                    self.player_hitted()
                    self.enemy_hitted(enemy)
                for bullet in self.bullets:                #我方发射的每个子弹
                    if enemy.colliderect(bullet):          #子弹是否和敌方战机碰撞
                        self.enemy_hitted(enemy)
                        self.bullets.remove(bullet)
                        self.remove_sprite(bullet)

            for bullet in self.bullets_enemy:              #敌方发射的每个子弹
                player = self.player
                if player and player.colliderect(bullet): #子弹是否击中我方战机
                    self.player_hitted()
                    self.bullets_enemy.remove(bullet)
                    self.remove_sprite(bullet)

        #我方战机被击中
        def player_hitted(self):
            player = self.player
            player.lives-=1
            self.score-=1
            if player.is_dead():  #如果销毁，则播放爆炸动画和声音
                #player_explosion_sound = self.resource.sounds['explosion_sound']
                explosion = Explosion(player.destroy_images, player.pos,player.crash_sound)
         #player_explosion_sound)
                explosion.sound_explosion.play()
                self.add_sprite(explosion)
                self.running = STOP
            else:    #播放被射中的声音
                player_hited_sound =  mixer.Sound(self.resource.sounds['player_hited_sound'])
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                player_hited_sound.play()

        #敌方战机被射中
            def enemy_hitted(self,enemy):
                enemy.lives -= 1
                self.score +=1
                if enemy.is_dead():    #如果销毁，则播放爆炸动画和声音
                    explosion = Explosion(enemy.destroy_images, enemy.pos,enemy.crash_sound)
                    explosion.sound_explosion.play()
                    self.add_sprite(explosion)
                    self.enemys.remove(enemy)
                    self.remove_sprite(enemy)

        game = SpaceInvander4(WIDTH,HEIGHT)
        game.run()
```

运行上述程序，可以看到战机可被子弹射中，爆炸而销毁，效果如图 5-17 所示，现在就可以检测战机和子弹的碰撞了，当一个战机的生命值为“0”时就会发生爆炸而销毁。限于篇幅，对于战机被击中、战机损坏等图片读者可自行添加。另外，战机、子弹等超出窗口怎么处理呢？这些都留待读者自己解决。

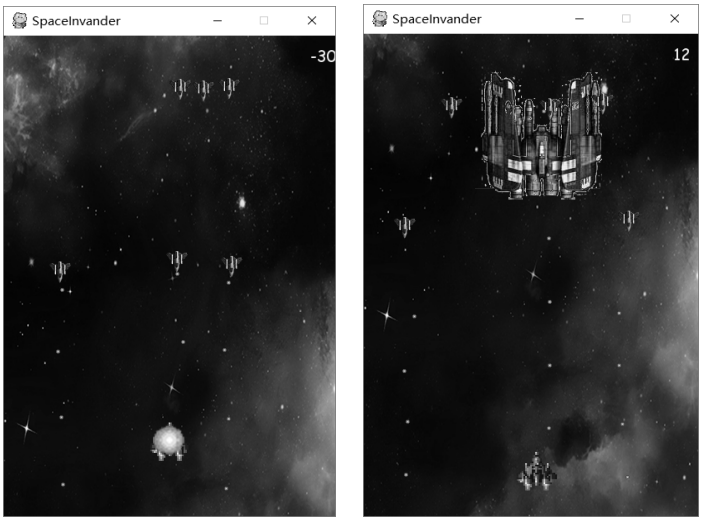


图 5-17　程序运行效果

5.7　习题

1．下列代码的输出结果是什么？（　　）

```
class C:
  def Print(self, line='Python'):
    print(line)

a = C()
a.Print('Java')
```

2．Python 的函数__init__()的作用是（　　）。

A．为使用一个类而初始化　　B．对象实例化时调用的函数

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

C．调用时，初始化数据属性的值为“0”　　D．上面的说法都不对

3．下列代码的输出是(　　)。

```
class Point:
   def __init__(self, x = 0, y = 0):
     self.x = x+1
     self.y = y+1

p1 = Point()
print(p1.x, p1.y)
```

A．0 0　　B．1 1　　C．None None　　D．x,y

4．解释下列代码中属性 name、surname 和 profession 之间的区别，并说明不同实例中这些属性的可能值。

```
class Smith:
 surname = "Smith"
 profession = "smith"

 def __init__(self, name, profession=None):
    self.name = name
    if profession is not None:
       self.profession = profession
```


5．对于下面表示二维平面上的点的类 Point，请完善函数__str__()，使其对一个 Point 变量 p 在函数 print(p) 中产生形如(x,y)的输出。另外，重载定义运算符+、–、<、==，使得对两个点 p、q，可直接使用 p+q 或 p–q 得到一个新的 Point 对象，并可以用 p<q 或 p==q 判断它们的大小或是否相等。

```
class Point:
   def __init__(self, x = 0, y = 0):
      self.x = x
      self.y = y

   def __str__(self):
      #补充你的代码
      #...
```

6．下列代码的输出是什么？(　　)

```
class A():
   def disp(self):
      print("A disp()")
class B(A):
   pass
obj = B()
obj.disp()
```

A．非法的继承语法　　B．创建对象错误，缺少参数

C．什么也不输出　　D．A disp()

7．下列代码是否正确？如果正确，那么输出是什么？(　　)

```
class A:
   def __init__(self, x= 1):
      self.x = x
class der(A):
   def __init__(self,y = 2):
      super().__init__()
      self.y = y
def main():
   obj = der()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        print(obj.x, obj.y)
```

A．错，调用方法的语法错了　　B．程序没任何错误，但不输出任何信息

C．1 0　　D．1 2

8．如果一个类从两个不同的类派生，则称为(　　)。

A．多层继承　　B．多继承　　C．层次继承　　D．Python 继承

9．下列程序的输出是什么？(　　)

```
class C1:
    def __init__(self):
        self.x = 1
class C2(C1):
    def __init__(self):
        super().__init__()
        self.x = 2
class C3(C2):
    def __init__(self):
        super(C3,self).__init__()
        self.x = 3

a = C1()
b = C2()
c = C3()
print(a.x,b.x,c.x)
```


10．下列程序的输出是什么？(　　)

```
class C1:
    def __init__(self):
        print('C1')
    def f(self):
        print('f in C1')

class C2:
    def __init__(self):
        print('C2')
        super().__init__()
    def f(self):
        print('f in C2')

class C3(C1, C2):
    def __init__(self):
        print('C3')
        super(C3,self).__init__()
    def g(self):
        print('g in C3')
        print('call f in C2')
        C2.f(self)

c = C3()
c.f()
c.g()
```

11．模仿下面的类 circle(圆)的代码，完善表示二维几何图形的类 Polygon(多边形)、Rectangle(矩形)、Square(正方形)、Triangle(三角形)。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
class Shape:
  def paint(self): pass

class Circle(Shape):
  def __init__(self, x, y, r):
    self.x = x
    self.y = y
    self.r = r

  def paint(self):
    print("画半径是{}、圆形在({},{})的圆",.format(self.r,self.x,self.y))

class Polygon(Shape):
  #...
class Rectangle(Polygon):
  #...
class Square(Rectangle):
  #...
class Triangle (Polygon):
  #...
```


12. 定义一个类 Person，其中包含 name(姓名)、sex(性别)、age(年龄)、address(住址)、email(电子邮箱)、phone(电话号码)等信息，然后在此基础上定义 Student(学生)、Staff(员工)，并在 Staff 基础上定义 Teacher(教师)。要求：Student 有一个 dict 属性记录其每门课程的学分，Staff 至少有一个 salary 属性表示其薪水，Teacher 有一个 list 对象存储其能教授的所有课程。在这些类的基础上编写一个简单的学校人员管理程序，包含输入、查找、修改、删除、显示等操作。
13. 实现二叉树的操作，如求一个二叉树的叶子结点数目、复制一颗二叉树、判断二叉树是否相等或判断是否具有相似结构，等等。
14. 在类 BiTreeNode 的基础上，实现一个较为完善的二叉搜索树类 BST，对其进行插入、查找、删除、修改、打印等操作，这些操作都作为类 BST 的方法而不是外部函数实现。请进一步实现一个二叉平衡树(选做题)。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 6 章　输入/输出

6.1　标准输入/输出

6.1.1　标准输出函数 print()

Python 内置的标准输出函数 print()可以接收多个输出值，这些值之间以逗号隔开，函数 print()请输出的值以空格隔开，并且在输出这些值后还会换到新的一行。例如：

```
print(2,3.14,'hello',[5,8.6])
```

输出：

```
2 3.14 hello [5, 8.6]
```

可以通过给函数 print()传递关键字参数 sep，以改变输出时输出项之间的分割字符串。

```
print(2,3.14,'hello',[5,8.6],sep = '-/')
```

输出：

```
2-/3.14-/hello-/[5, 8.6]
```

函数 print()默认在输出的最后输出一个换行符“\n”，即换到新的一行。

```
for i in range(3):
    print(i)
```

输出：

```
0
1
2
```

若要改变函数 print()的默认换行操作，则可以给函数 print()的关键字参数 end 传递一个相应的值。例如：

```
for i in range(3):
    print(i,end='+-')
```

上述代码输出字符串“+-”作为每一个函数 print()的结束，程序的输出结果为：

```
0+-1+-2+-
```

6.1.2　格式化输出

如果想对输出做更多的格式控制，而不是简单地使用空格等符号分隔输出的值，则其中一种方法是使用 C 语言风格的格式串控制输出的格式。格式转换符和输出项的对应关系如图 6-1 所示。

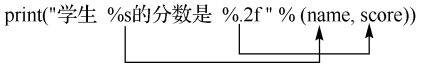


图 6-1　格式转换符和输出项的对应关系

其中，引号包围的"学生 %s 的分数是 %.2f"是一个**格式字符串**(简称“**格式串**”)，格式串后面是以%开头用圆括号括起来的一些输出项。格式串包含了格式转换符，如%s 和%d，用于表示对应的输出项以什么格式输出。其中，%s 表示格式串后的输出项 name 以字符串形式输出，而%.2f 表示输出的浮点数

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

用四舍五入保留小数点后两位。格式转换符和输出项是逐一对应的。

常见的格式转换符有：

- %d：整数；
- %f：浮点数；
- %s：字符串；
- %p：数据的内存地址(十六进制)。

例如，执行下面的代码：

```
name = 'Wang'
score = 56.345
print("学生 %s 的分数是 %.2f" %(name, score))
print("该学生的学号是 %d ." %3)
```

输出结果：

```
学生 Wang 的分数是 56.34
该学生的学号是 3 .
```

除直接以 C 语言风格格式串控制输出的格式外，还可以借助字符串的处理函数或方法对一个字符串进行处理，以产生一个格式化的新字符串。通常，可以使用前面介绍过的 str 的 format()方法对一个字符串格式化。通过{}和:来代替传统的%方式。例如：


```
name = 'Wang'
score = 56.345
print("学生 {}的分数是 {:.2f}".format(name, score))
print("该学生的学号是 {}.".format(3))
```

输出：

```
学生 Wang 的分数是 56.34
该学生的学号是 3 .
```

str 的 format()方法针对一个包含占位符{}的字符串，将一些对象传给 format()方法来取代占位符{}。

除 format()方法外，还可以使用其他的一些方法或函数对字符串进行处理，以产生一个格式化的新字符串。例如，可以用 str.rjust()、str.ljust()、str.center()控制字符串输出的宽度和字符串的对齐方式(靠右、靠左、中间)。例如：

```
s = 'hi'
t = 'the'
print(s.rjust(5),t.rjust(10))
print(s.ljust(5),t.ljust(10))
print(s.center(5),t.center(10))
s.rjust(5)
```

输出：

```
   hi       the
hi    the
  hi     the
'   hi'
```

s.rjust(5)表示产生一个新的宽度是 5 的字符串，并且'hi'在这个字符串中是靠右对齐的，左边不足的部分用空格填充。最后输出的是字符串'   hi'。

在输出或产生格式化字符的过程中，为了将非 str 字符串类型的值转化成 str 类型的值，则可

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

以使用函数 str()或 repr()，前者将值转化为人可以阅读的字符串，而后者将值转化为可供解释器读取的形式，如果某对象没有适于人阅读的解释形式，则函数 str()会返回与函数 repr()等同的值。例如：

```
print(str('123'))
print(str(123))
print(repr('123'))
print(repr(123))
```

输出：

```
123
123
'123'
123
```

可以看出，函数 str()返回的是一个通常的字符串对象，而函数 repr()返回的是解释器内部的表示形式。

6.1.3 美观输出函数 pprint()

pprint 模块包含一个“漂亮的打印机”，用于生成美观的数据结构视图，即生成可以被解释器正确解析的、可以轻松阅读的数据结构表示。例如，尽量打印在一行，多行打印时缩进，以递归方式支持漂亮打印“列表、元组和字典”，在打印字典前会按键对字典进行排序等。该功能虽然简单但是很有用，尤其是在调试数据结构时。

1．函数 pprint()

使用 pprint 模块的最简单方法是通过函数 pprint()。该函数以单行格式输出对象，如果不符合允许的宽度，则将对象分成多行。例如，下面代码中的 data 引用的 list 对象的前两个元素以单行形式输出，而第三个元素则以多行形式输出。

```
data = [
    (1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
    (2, ['hello', 3.14]),
    [[1,2,3],{4,5,6},(7,8,9,10),['e','f','g'],{'h','i'},{'j','k','l'}],
     ]
from pprint import pprint
print('print()函数输出:')
print(data)
print('pprint()函数输出')
pprint(data)
```

以上代码的输出如下：

```
print()函数输出:
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}), (2, ['hello', 3.14]), [[1, 2,
3], {4, 5, 6}, (7, 8, 9, 10), ['e', 'f', 'g'], {'i', 'h'}, {'j', 'l', 'k'}]]
pprint()函数输出
[(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
 (2, ['hello', 3.14]),
 [[1, 2, 3],
  {4, 5, 6},
  (7, 8, 9, 10),
  ['e', 'f', 'g'],
  {'i', 'h'},
  {'j', 'l', 'k'}]]
```

由此可见，函数 pprint()的输出格式更加美观。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

对于以下问题，如非常深的数据结构，不希望输出所有细节的数据，数据格式不正确，格式化文本太大而无法管理，某些数据可能无关紧要等，此时，可以通过为函数 pprint() 设置 depth 参数值，以控制其输出的嵌入深度。例如：

```
pprint(data,depth=1)
print("depth=2")
pprint(data,depth=2)
```

输出：

```
[(...), (...), [...]]
depth=2
[(1, {...}),
 (2, [...]),
 [[...], {4, 5, 6}, (...), [...], {'i', 'h'}, {'j', 'l', 'k'}]]
```

格式化文本的默认输出宽度为 80 列。如果要调整该宽度，可设置函数 pprint() 的 width 参数；如果要调整缩进，则可设置函数 pprint() 的 indent 参数。例如：

```
pprint(data,width=10,indent=4)
```

输出：

```
[   (   1,
        {   'a': 'A',
            'b': 'B',
            'c': 'C',
            'd': 'D'}),
    (   2,
        [   'hello',
            3.14]),
    [   [   1,
            2,
            3],
        {   4,
            5,
            6},
        (   7,
            8,
            9,
            10),
        [   'e',
            'f',
            'g'],
        {   'h',
            'i'},
        {   'j',
            'k',
            'l'}]]
```

如果一个用户定义类的内部定义了__repr __() 方法，则函数 pprint() 使用的类 PrettyPrinter 也可以支持这种自定义类的输出。例如，下列表示文件夹的类 Folder 定义了这个方法，所以函数 pprint() 可以输出该类的对象：

```
from pprint import pprint
class Folder:
    def __init__(self, name, contents=[]):
        self.name = name
        self.contents = contents[:]
    def __repr__(self):
        return (
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                'Folder(' + repr(self.name) + ', ' +
                repr(self.contents) + ')'
            )
    folders = [
        Folder('Python'),
        Folder('Music', [Folder('流行歌曲'),Folder('古典歌曲')]),
        Folder('Video', [Folder('欧美影视'),Folder('国内影视'),Folder('印度影视')]),
    ]
    pprint(floders)
```

输出：

```
    [Folder('Python', []),
    Folder('Music', [Folder('流行歌曲', []), Folder('古典歌曲', [])]),
    Folder('Video', [Folder('欧美影视', []), Folder('国内影视', []), Folder('印度影视', [])])]
```

2．函数 pformat()

如果要格式化数据结构而不将其直接输出到流中，则可使用函数 pformat()来构建对象的字符串表示。例如：

```
    from pprint import pformat
    s = pprint.pformat(data)
    for line in s.splitlines():
      print(line.rstrip())
```


上述代码将 data 首先用函数 pformat()格式化输出到一个字符串 s 中，然后循环打印 s 中用换行符\n 隔开的每一个字符串，输出结果如下：

```
    [(1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
     (2, ['hello', 3.14]),
     [[1, 2, 3],
      {4, 5, 6},
      (7, 8, 9, 10),
      ['e', 'f', 'g'],
      {'i', 'h'},
      {'j', 'l', 'k'}]]
```

注意

- 字符串的 splitlines()方法用换行符\n 分割字符串。
- rstrip()方法删除一个字符串右边的空格。同样，lstrip()方法删除左边的空格，strip()方法删除两边的空格。这些方法默认为删除空格，也可以传入需要从两边或其中一边删除的字符，如'a'就是删除字符串两边的字符“a”。

3．PrettyPrinter 类

可以创建 PrettyPrinter 漂亮打印机对象并传入格式化控制的参数来控制格式化输出。例如，下列代码在构造 PrettyPrinter 类时设置了 indent(缩进)参数：

```
    import pprint
    pp = pprint.PrettyPrinter(indent=6)          #创建缩进为 6 的 PrettyPrinter
    pp.pprint(data)
```

输出：

```
    [    (1, {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}),
         (2, ['hello', 3.14]),
         [    [1, 2, 3],
              {4, 5, 6},
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                (7, 8, 9, 10),
                ['e', 'f', 'g'],
                {'i', 'h'},
                {'j', 'l', 'k'}]]
```

当然，也可以定义继承 PrettyPrinter 类的派生类，对格式化输出进行更灵活的控制。在定义派生类时，要重载格式化方法_format()。例如：

```
from __future__ import division
import pprint
#将原来的pprint.PrettyPrinter保存在pprint.old_printer
if not hasattr(pprint,'old_printer'):
   pprint.old_printer=pprint.PrettyPrinter

class MyPrettyPrinter(pprint.old_printer):
   def _format(self,obj,*args,**kwargs): #重新定义_fromat()方法
      if isinstance(obj,float):
         obj=round(obj,7)
      return pprint.old_printer._format(self,obj,*args,**kwargs)

pprint.PrettyPrinter=MyPrettyPrinter #设置pprint.PrettyPrinter为新的MyPrettyPrinter

def pp(obj):
   pprint.pprint(obj)

if __name__=='__main__':
   pp(3/19)
```

输出：

```
0.1578947
```

由此可见，新的 My PrettyPrinter 的_format()方法只对第一个参数做处理，然后交给原来的 PrettyPrinter。

6.1.4 标准输入（内置函数 input()）

内置函数 input()的格式为：

```
input([prompt])
```

该函数从键盘中读取一个字符串，且该函数还可以带一个表示提示的字符串。无论输入的是什么类型的数据，该函数都返回一个字符串，程序根据需要将该函数返回的字符串转换成相应的类型，如 int、float。例如：

```
name = input("请输入姓名：")
print(name, type(name))
score = input("请输入分数：")
print(score, type(score))
score = float(score)    #用内置函数float()转换成float类型
print(score, type(score))
```

输出：

```
请输入姓名：李平
李平 <class 'str'>
请输入分数：67.8
67.8 <class 'str'>
67.8 <class 'float'>
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

6.2 文件读/写

程序中的对象是保存在计算机内存中的，一旦程序结束这些数据就没有了。为了永久保存数据，所有操作系统都会提供文件读/写的功能，以用于将数据保存到外部存储器上，如硬盘。在程序运行时，可以随时从外部存储器上的文件中将这些数据读入到计算机内存中进行处理。

文件操作基本分为三个步骤：

- 打开(open)一个文件；
- 读(read)/写(write)文件内容；
- 关闭(close)这个文件。

6.2.1 内置函数 open()

内置函数 open()用于打开一个文件。该函数接收一个文件名(文件路径)作为参数，返回一个**文件对象**(也称**句柄**)，然后就可以通过这个文件句柄读或修改(写)该文件的内容。

函数 open()的规范是：

```
file object = open(file_name [, access_mode][, buffering])
```


第一个参数 file_name 是要打开的文件名(文件路径)，后面是可选参数，参数 access_mode(**访问模式**)用一些字符表示使用文件的方式。例如，字符 r 表示该文件只能读(不能被修改)；字符 w 表示只能修改(写)这个文件，如果存在同名的文件，则会被擦除；字符 a 表示以“追加”方式向文件的末尾写入数据；字符 r+表示该文件既可以读也可以写。如果没有提供 access_mode 参数，则该参数默认值是 r。

默认情况下，文件以**文本模式**(**text mode**)打开，即读或写的数据都是特定编码的字符，如果没有指定编码，则字符编码默认采用操作系统的编码。可以在 access_mode 参数中添加字符 b 表示文件以**二进制模式**(**binary mode**)打开，即用最原始的字节(byte)读写数据。

可选参数 buffering 表示读/写文件时的缓冲区的大小，如果被设为 0，就不会有缓冲区。如果取负值，则缓冲区大小为系统默认。

当设置缓冲区后，读取的文件块的内容首先放入缓冲区，然后才转变为程序的数据。同样，程序的数据首先放入缓冲区，等缓冲区满后才真正写入文件。因为读/写外部设备上的数据的速度要比 CPU 处理数据速度慢得多，频繁地直接读/写外部设备上的数据会降低程序效率。如果将数据首先放在缓冲区的一块内存中，待缓冲区满时再一次性将大块数据读/写到外部设备，则可提高读/写数据的速度。

例如，以只读模式打开一个不存在的文件：

```
f = open('text.txt','r')
```

会产生“FileNotFoundError(文件未发现错误)”：

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)

<ipython-input-1-94b66ecee0a5> in <module>()
----> 1 f = open('text.txt','r')
FileNotFoundError: [Errno 2] No such file or directory: 'text.txt'
```

因为以 r(只读模式)打开文件，但不存在这个文件，所以出错。如果以 w(写模式)打开文件，则可以创建一个新文件或覆盖已经存在的文件。例如：

```
f = open('text.txt','w')
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

表 6-1 列出了文件打开模式的字符及其含义，这些字符表示的模式可以组合。例如，模式'wb'表示以二进制模式打开文件，并将文件清空为 0 字节；'rb'表示以二进制模式打开文件但不清空文件。

表 6-1　文件打开模式的字符及其含义

字　符	含　义	字　符	含　义
'r'	只读方式打开文件(默认)	'b'	二进制模式
'w'	只写方式打开文件，会首先清除原有文件内容	't'	文本模式（默认）
'x'	以独占创建方式打开一个文件，如果文件已经存在则失败	'+'	打开一个磁盘文件用于读/写
'a'	以写方式打开文件，新内容加到已有内容后面		

6.2.2　文件对象的方法

只要使用函数 open()创建一个文件对象，就可以通过文件对象的方法如 read()、readline()、readlines()、write()、writelines()等对这个文件对象代表的文件进行各种读/写操作。

1．write()方法

write()方法的格式：

```
fileObject.write(str)
```

向文件对象 fileObject 代表的文件中写入一个字符串 str，并返回写入的字符个数。

例如，调用 write()方法向新打开的文件对象 f 表示的文件里写入一个字符串，然后调用 close()方法关闭它：

```
count = f.write('Hello, world!')
print('写入的字符个数:',count)
f.close()
```

输出：

```
写入的字符个数: 13
```

如果在执行这个程序的当前目录下打开这个文件(test.tx)，就可以看到如图 6-2 所示的文件内容。

图 6-2　test.tx 的文件内容

对于已经存在的文件可以用只读模式('r')打开这个文件，然后可以用 read()方法读取文件的全部内容。例如：

```
f = open('text.txt','r')
f.read()
```

输出：

```
'Hello, world!'
```

每次读/写完文件，就要及时调用函数 close()关闭它，否则容易造成数据丢失或其他程序无法读/写的错误。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

函数 close()的语法格式如下。

```
f.close()
```

可以使用 **with...as** 打开文件，以保证文件的函数 close()被自动调用，从而可以防止因忘记调用函数 close()而引起的问题。例如：

```
with open('text.txt','w') as f:
    f.write("教小白精通编程")
with open('text.txt','r') as f:
    print(f.read())
```

输出：

```
教小白精通编程
```

可以看出，文件内容被新写入的内容替换了，如何在文件中已有内容的基础上添加新的内容呢？办法是在调用函数 open()时以“追加模式”，即'a'打开文件。例如：

```
#以'a'，即"追加模式"打开文件
with open('text.txt','a') as f:
    f.write("新的内容")

#再一次读入文件内容
with open('text.txt','r') as f:
    print(f.read())
```


输出：

```
教小白精通编程新的内容
```

2．read()方法

read()方法的格式如下：

```
fileObject.read([size])
```

其意义是，从文件对象 fileObject 表示的文件中读取 size 个字符(文本模式)或 size 字节(二进制模式)。如果没有提供默认参数 size，则 read()方法会一次读入文件的所有内容，如果文件很大，那么这种方法就不行了，应该传入一个参数 size，每次最多读取 size 字节。

该方法返回读取的字符串(或字节序列)，如果遇到了文件结尾，则返回一个空串。例如：

```
with open('text.txt','r') as f:
    while True:
        chunk = f.read(4)   #每次读取 4 个字符
        if not chunk:
            break
        print(chunk)
```

输出：

```
教小白精
通编程新
的内容
```

3．readline()方法和 readlines()方法

readline()方法的格式如下：

```
fileObject.readline([size])
```

其意义是，从文件对象 fileObject 表示的文件中读取一行，即从文件对象的当前位置一直读取内容直到遇到换行符\n，或者如果提供了参数 size，则最多读取 size 个字符。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

该方法返回读取的一行字符串(包括换行符\n)，遇到文件结尾时，返回的是空串。但是，对于文件中的空行，则返回只有换行符\n 的字符串。例如：

```
with open('text.txt','a') as f:
    f.write("Hello, World!")
    f.write("\n 你好")
    f.write("\nhello")

with open('text.txt','r') as f:
    while True:
        line = f.readline()
        if not line:
            break
        print(line.strip()) #把末尾的'\n'删掉

print()
```

输出：

```
Hello, world!
你好
hello
```

读取所有行的 **readlines**()方法的格式如下：

```
fileObject.readlines([sizehint])
```

读取文件直到结束符 EOF，并返回一个列表。如果提供了参数 sizehint，那么读取的所有行的字节数不超过 sizehint。

```
with open('text.txt','r') as f:
    for line in f.readlines():
        print(line.strip()) #把末尾的'\n'删掉
```

输出：

```
Hello, world!
你好
hello
```

因为 readlines()方法返回的是由读取的行构成的一个 list 对象，所以可以用处理 list 对象的函数或方法对它进行操作。例如，可以用函数 enumerate()得到每一行的索引：

```
with open('text.txt','r') as f:
    for index, line in enumerate(f.readlines()):
        print(index,line.strip())
```

输出：

```
0 Hello, world!
1 你好
2 hello
```

还可以用 for 循环访问文件的每一行。例如：

```
with open('text.txt','r') as f:
    for line in f:
        print(line, end='')
```

输出：

```
Hello, world!
你好
hello
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

4．writelines()方法

writelines()方法的格式如下：

```
fileObject.writelines(iterable)
```

将一个 iterable 对象表示的一系列字符串写入文件对象 fileObject 代表的文件中。注意，write()方法一次只能写一个字符串，而 writelines()方法一次可以写多个字符串到一个文件中，且该方法没有返回值。

例如：

```
f = open(r'test.txt', 'w')
f.writelines(['hello world', '教小白精通编程'])
f.close()
f = open(r'test.txt')
f.read()
```

输出：

```
' hello world教小白精通编程'
```

请读者思考：“如何使每个字符串作为文件的一行？”

6.2.3 二进制文件读/写

前面讲的都是读取文本文件，如果要读写二进制文件，如图片、视频等，则可以在访问模式如'r'后面增加一个二进制模式'b'。例如，用'rb'模式表示以二进制模式打开文件：

```
with open('test.bin', 'wb') as f:
    f.write(b'hello world')

with open('test.bin', 'rb') as f:
    data = f.read()
    print(type(data))
    print(data)
```

输出：

```
<class 'bytes'>
b'hello world'
```

注意

- 对于二进制模式，不能用 readline()、readlines()、writerlines()这类按行的读/写方法。
- 读/写的二进制数据都是字节形式，即 f.read()返回的是一个字节串(bytes)而不是字符串。因此，如果想要读/写文本数据，则必须将写入文件中的数据编码为字节字符串，或必须将从文件读取的数据解码为文本字符。可以通过 str 的 encode()方法和 decode()方法传入合适的编码名。例如:

```
with open('test2.bin', 'wb') as f:
    text = '教小白精通编程'
    f.write(text.encode('utf-8'))

with open('test2.bin', 'rb') as f:
    data = f.read()
    print(type(data))
    print(data.decode('utf-8'))
```

输出:

```
<class 'bytes'>
教小白精通编程
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

6.2.4 tell()方法和 seek()方法

有时不希望总是从文件开头到文件结尾这样读写文件，而希望定位到文件的某个位置进行读/写，这就需要用到 tell()方法和 seek()方法。

file 对象的 tell()方法返回文件对象当前所处的位置，它是从文件开头开始算起的字节数。

file 对象的 seek()方法用于定位到文件的某个位置。其格式是:

```
f.seek(offset, from_what)
```

其意义是，将文件定位到一个参考点加上偏移量 offset 的位置。参考点由 from_what 决定。from_what 的值如果是 0，则表示开头位置；如果是 1，则表示当前位置；如果是 2，则表示文件的结尾。例如：

- seek(x,0)：从开头位置，即文件首行首字符开始往后偏移 x 个字符或字节。
- seek(x,1)：从当前位置往后偏移 x 个字符或字节。
- seek(–x,2)：从文件的结尾往前偏移 x 个字符或字节。

注意

对于非二进制访问模式，只允许从开头进行偏移！例如:

```
f = open('abc.txt', 'wb+')
f.write(b'abcdefg1234567')

f.seek(5)              #移动到文件的第 6 字节
print(f.read(1))       #读取 1 字节
f.seek(-5, 2)          #移动到文件的倒数第 5 字节
print(f.read(3))       #读取 3 字节
```

输出:

```
b'f'
b'345'
```

总结

介绍标准输入函数 input()和输出函数 print()。可通过 C 语言风格的%格式串或生成一个格式化新字符串控制 print()的输出格式。

介绍字符串 str 的格式化方式：一种方式是利用 str 的 format()方法；另一种方式是利用字符串的各种切割、连接等方法。

介绍 pprint 模块的 pprint()格式化打印函数、pformat()格式化函数和 PrettyPrinter 类。

内置函数 open()用于打开或创建文件，该函数返回一个文件对象。文件访问模式是由几个字符，如'r' 'w' 'a' 'b' '+'组合表示的。文件分为二进制和文本文件。

通过文件对象的 read()方法、write()方法等可以读/写文件内容，通过 tell()方法、seek()方法可以查询或改变当前位置。

文件使用完，要用 close()方法关闭。也可以采用 with ... as 打开文件，以保证文件总能自动被关闭。

6.3 习题

1. 函数 open()中的文件模式参数'r'和'a'的含义是什么?(　　)

A. 读、追加　　B. 追加、读　　C. A 和 B 都对　　D. A 和 B 都不对

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2．二进制读/写模式是哪一个？（　　）

A．wb+　　B．w　　C．wb　　D．w+　　E．上述都不对

3．下面哪句是正确的？（　　）

A．当以读模式打开一个不存在文件时，会出现错误。

B．当以写模式打开一个不存在文件时，会创建一个新文件。

C．当以写模式打开一个存在文件时，存在的文件会被新文件替换。

D．上述都对。

4．下列代码的结果是什么?（　　）

```
f = open("test.txt")
```

A．以读/写模式打开 test.txt。　　B．以读模式打开 test.txt。

C．以写模式打开 test.txt。　　D．以读或写模式打开 test.txt。

5．编写一个程序找出一个文件中最长的单词。

6．编写一个程序，从键盘上输入一组学生的信息(假如每个学生有姓名、学号和分数这三个数据项)，要求每个学生的信息在一行中输入，将所有学生的信息写入一个文本文件中，再从该文件中读取并显示所有学生的信息，要求每个数据项占据固定的宽度并左对齐。修改第三个学生的信息，并将修改后的学生信息写入文件中，再次读入并显示所有学生数据，以验证数据是否被正确修改了。

7．编写一个程序，从键盘输入一组三维坐标点数据，将这些数据写入一个二进制文件中，然后单独读取第三个坐标点的数据，修改后再次写入文件中。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 7 章　错误和异常

7.1　错误

编写程序时不可避免地会产生各种错误，错误主要有两种，语法错误(syntax errors)和运行时错误(run-time error)。运行时错误是程序运行过程中出现的错误，又分为异常错误和逻辑错误。语法错误和异常错误都会使程序停止执行，而逻辑错误不会使程序停止执行，但运行结果和预期结果不一致。例如，一个判断年份是否为闰年的程序，对于某非闰年(如 1999 年)却判断为闰年。

7.1.1　语法错误

日常生活中所说的“你饭了吗？”这句话并不符合日常语言的语法，即存在“缺少动词”的语法错误，正确的说法应该是“你吃饭了吗？”。同样，Python 程序代码如果不符合 Python 语言的语法也会导致“语法错误”。例如：

```
score = 90.5
if score < 60
  print(' 不及格')
```

将产生语法错误(SyntaxError)：

```
File "<ipython-input-3-77e947c3f049>", line 2
  if score < 60
                ^
SyntaxError: invalid syntax
```

Python 解释器在执行 if score < 60 语句时会立即提示“语法错误：无效的语法(SyntaxError: invalid syntax)”，并且给出了出错的代码语句。这个错误是因为少了一个冒号：引起的。语法错误是初学者遇到最多的错误，这是因为他们对语言的掌握并不熟练，随着熟练程度的提高，这种错误很容易纠正，并且会越来越少。再如：

```
score = 90.5
If score < 60:
 print(' 不及格')
```

也产生了语法错误：

```
  File "<ipython-input-2-83cc656198aa>", line 2
    If score < 60:
          ^
SyntaxError: invalid syntax
```

这个语法错误是因为将关键字 if 写成了 If。又如：

```
score = 90.5
if score < 60:
print(' 不及格')
```

产生了“缩进错误(IndentationError)”：

```
  File "<ipython-input-1-18eeed607578>", line 3
    print(' 不及格')
      ^
IndentationError: expected an indented block
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

这是一个 IndentationError: expected an indented block(缩进错误：期望一个缩进块)的语法错误。IndentationError 是从语法错误基类 SyntaxError 派生的一个派生类。例如：

```
issubclass(IndentationError, SyntaxError)
```

输出：

```
True
```

Python 常见语法错误有：

- 不正确的缩进；
- 关键字遗漏或拼错(如将 elif 写成了 else if，将 True 写成了 true，将 if 写成了 If 等)；
- 漏掉了一个符号，如冒号、逗号或括号；
- 语法上需要语句的地方缺少语句，如空的 if 语句缺少了 pass 空语句 。

Python 解释器会告诉用户出现语法错误的所在行，通常，只要观察该行及其上一行就能找到语法错误。但是，有时，一个语法错误要在很多行以后才显示出来，这时就需要在出现语法错误的前面的代码行里去寻找引起相关错误的原因，不过，这种情形很少发生。

7.1.2 运行时错误：异常

另外一种错误是从语法上看程序本身没有错误，但是运行过程中会出错，这种运行时错误称为异常。例如：

```
10*(1/0)
```

上述语句在语法上没有任何问题，但执行上述语句会产生一个异常“ZeroDivisionError(除 0 错误)”。这是因为“0”不能作为除数。再如：

```
if score<60:
   print(' 不及格')
```

产生了名字错误(NameError)：

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-5-2181deb54d75> in <module>()
----> 1 if score<60:
      2    print(' 不及格')
NameError: name 'score' is not defined
```

上述代码在语法上也没有任何问题，但执行时同样抛出了一个“NameError(名字错误)”的异常，这是因为，代码中并没有说明 score 这个名字是做什么用的。

再如：

```
import Math
```

同样产生一个叫作 ModuleNotFoundError 的错误或异常。

下列代码产生一个“TypeError(类型错误)”的异常，这是因为 str 字符串对象和 int 对象不能直接执行加法运算：

```
'2' + 2
```

产生错误：

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-d2b23a1db757> in <module>()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
----> 1 '2' + 2

TypeError: must be str, not int
```

Python 定义了很多不同的错误或异常类，用于表示不同类型的错误，如“除 0 错误(ZeroDivisionError)”“文件未找到(FileNotFoundError)”“导入错误(ImportError)”等。所有内置的错误或异常类都是从一个最基本的异常类 BaseException 派生出来的。大多数异常类是从 BaseException 类的派生类 Exception 派生出来的。ZeroDivisionError 类、NameError 类、TypeError 类、FileNotFoundError 类、ModuleNotFoundError 类等都是 Exception 类的子类，这些异常类形成一个层次性的派生关系，前面的语法错误也是从 Exception 类派生出来的。实际上，ModuleNotFoundError 类的直接基类是 ImportError 类，而 ImportError 类的直接基类是 Exception 类。因此，Exception 类是 ModuleNotFoundError 类的间接基类。例如：

```
print(issubclass(SyntaxError, Exception))
print(issubclass(ZeroDivisionError, Exception))
print(issubclass(NameError, Exception))
print(issubclass(TypeError, Exception))
print(issubclass(ModuleNotFoundError, ImportError))
print(issubclass(ImportError, Exception))
```

输出：

```
True
True
True
True
True
True
```

这些错误异常的派生关系可以参考官方文档：https://docs.python.org/3/library/exceptions.html。

出现运行时异常时，Python 除输出异常的类型及其解释外，还会在错误信息的前面打印引起异常的 Traceback(跟踪栈)，这个跟踪栈给出了引起错误异常的函数调用关系，通过这个跟踪栈提供的回溯跟踪信息，程序员可了解引起异常的语句之间的调用关系。

常见的运行时异常有：

- 除 0 错误；
- 对不兼容的类型执行操作；
- 使用尚未定义的标识符；
- 访问不存在的列表元素、字典值或对象属性；
- 试图访问不存在的文件。

7.1.3 逻辑错误

有时，程序既没有语法错误，也没有抛出运行时异常错误，即程序能正常执行，但是运行的结果和预期的结果不一致。例如，要计算 1 到 10 之间的整数之和，但是结果输出不是 55 而是其他数值，这种错误是因为算法存在内在的逻辑问题造成的，这种运行时错误称为“**逻辑错误**”。

常见的逻辑错误如下。

- 使用错误的变量名称。例如，在类的方法中，访问类对象的属性(变量或方法)时忘记写 self. 前缀，或者在一个函数中访问全局变量时忘记声明这个全局变量等。
- 将块缩进到错误的级别。
- 使用整数除法而不是浮点除法。
- 运算符优先级错误。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- 布尔表达式出错。
- 其他数值错误。

7.2 异常处理

一旦程序出现了异常，将会终止当前函数，并将这个异常传递给调用该函数的函数，这个过程会一直进行下去，直到有一个函数处理了这个异常。例如，函数 A 调用了函数 B，函数 B 调用了函数 C，而函数 C 中某处出现了一个异常，函数 C 如果没有处理，函数 C 就会停止执行并将异常回传给函数 B，如果函数 B 仍然没有处理，函数 B 就会停止执行并将异常回传给函数 A，如果最终没有一个进程处理这个异常，则程序最终会停止，并打印输出 Traceback 回溯堆栈。程序员可根据这个 Traceback 回溯堆栈找到引起异常的最初的那个函数。

编写程序的时候，如果知道某段代码可能会导致某种异常，而又不希望程序以堆栈跟踪的形式终止，就需要编写**异常处理**的程序代码。可以通过 **try/except** 或 **try/finally** 语句(或者它们的组合)对出现的异常进行处理。如果异常得到了处理，那么程序就可以继续运行下去。异常处理可以采用以下形式中的其中一种：

```
try...except...
try...except...else...
try...except...else...finally...
try...except...except...else...finally...
try...finally...
```


7.2.1 捕捉异常的基本形式

捕捉异常的基本形式是：

```
try...except 程序块
```

将可能引起异常的代码放在 **try** 子句(程序块)中，如果 **try** 子句完成后没有异常发生，就会忽略 **except** 程序块(也称 **except 子句**)继续执行后续代码。如果 **try** 子句中代码执行时出现了错误而抛出异常时，**except** 程序块(**子句**)会处理出现的**异常**(错误)。一旦异常被 except 子句处理完，程序就会从“try…except”语句后面的语句继续执行下去。

except 子句对异常的具体处理，既可能是向用户报告某种发生的错误，也可能是终止程序的执行，还可能继续将异常交给其上一级调用函数处理，甚至也可能什么也不做。

看下面的例子：

```
try:
    x = int(input('Enter x: '))
    y = int(input('Enter y: '))
    print(x/y)
    print('x/y 结果正常')
except:
    print('出现了异常! ')
print('hello world')
```

执行这段代码：

```
Enter x: 3
Enter y: 0
出现了异常!
hello world
```

当输入的 y 为 0 时，print(x/y)的 x/y 抛出异常，try 程序块里的后续语句不会被执行，Python

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

会寻找能够处理这个异常的 except 子句(异常处理子句)，这个 except 子句输出了一个字符串信息“出现了异常！”，当这个异常被 except 子句处理完后，程序继续执行“try…except”语句后面的语句，即语句“print('hello world')”。

7.2.2 捕获特定类型的异常

上面的 except 后面直接是一个冒号:，说明该 except 子句可以捕获 try 块里抛出的任何类型的异常。有时，可能需要根据不同类型的异常做不同的异常处理，即在 except 关键字后面说明具体的异常类型，以捕获这个特定异常类型的异常对象，可以用 as 关键字给这个类型的异常对象起一个名字。例如：

```
try:
    x = int(input('Enter x: '))
    y = int(input('Enter y: '))
    print(x/y)
    print('x/y 结果正常')
except ZeroDivisionError as e:          #处理 ZeroDivisionError 异常
    print('x/y 出现了除 0 的错误！',e)
except ValueError  as e:                #处理 ValueError 异常
    print('类型错误 ValueError！',e)
print('hello world')
```

执行这段代码：

```
Enter x: 3
Enter y: a
类型错误 ValueError！ invalid literal for int() with base 10: 'a'
hello world
```

以上程序在执行 y = int(input('Enter y: '))时将输入的字母 a 转换为 int 类型值时产生了错误，当用户输入的不是一个整数而是一个字符时，类型转换函数 int()就会产生 ValueError 异常。try 子句后面有两个 except 子句，可分别捕获处理 ZeroDivisionError 和 ValueError 异常。因此，这个异常被 ValueError 类的异常处理 except 子句捕获，该异常处理子句只是输出了这个异常对象的信息：“类型错误 ValueError！ invalid literal for int() with base 10: 'a'”。如果这个异常得到了处理，那么 Python 解释器就会继续执行下一条语句“print('hello world')”。

try 子句中可能会抛出不同类型的异常，如果想针对每种异常做针对性的异常处理，则可以用多个 except 子句。当 try 子句抛出一个异常时，就根据该异常的类型按照从上往下的次序寻找匹配的异常处理子句。因此，特殊的异常应该在前面，而更一般的异常放在后面。这是因为，如果更一般的异常先被处理后，后面的特殊异常就不会被处理了。例如，假设 ExceptA 是从 ExceptB 派生出来的异常类型。如果按照如下次序捕获 ExceptA 和 ExceptB：

```
…
except ExceptB:
   #处理 ExceptB 的异常
except ExceptA:
   #处理 ExceptA 的异常
…
```

则“except ExceptA:”子句永远不会被处理，因为 ExceptA 的异常对象也是 ExceptB 的异常对象，会首先被“except ExceptB:”子句捕获。

7.2.3 捕获未知的内置异常

然而，在实际情况中可能很难预知所有可能出现的异常的类型，如果想处理那些未知类型的异

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

常，则可以在捕获特殊的异常的后面再捕获更一般的异常，由于 Python 的所有异常的类型都是从 BaseException 这个异常的基类派生出来的，因此可以用最后一个 except 子句捕获这个 BaseException 异常，也就能够捕获所有的内置异常类型对象了。

注意

由于 Exception 是大部分异常的父类，通常情况下也可用 Exception 代替 BaseException。例如：

```
try:
    x = int(input('Enter x: '))
    y = int(input('Enter y: '))
    print(x/y)
    print('x/y 结果正常')
except ZeroDivisionError as e:          #处理 ZeroDivisionError 异常
    print('x/y 出现了除 0 的错误! ',e)
except ValueError  as e:                #处理 ValueError  异常
    print('类型错误 ValueError! ',e)
except BaseException  as e:
     print('BaseException 异常处理! ',e)

print('hello world')
```


输出：

```
Enter x: 3
Enter y: er
类型错误 ValueError!  invalid literal for int() with base 10: 'er'
hello world
```

7.2.4 else 子句

可以在最后的 except 子句的后面添加一个 **else** 子句，当没有异常发生时，会自动执行 else 子句。但如果发生了异常，则不会执行这个 else 子句。例如：

```
try:
    x = int(input('Enter x: '))
    y = int(input('Enter y: '))
    print(x/y)
    z = int(input('Enter z: '))
    print(x/z)
except ZeroDivisionError as e:              #处理 ZeroDivisionError 异常
    print('x/y 出现了除 0 的错误! ',e)
except ValueError  as e:                    #处理 ValueError  异常
    print('类型错误 ValueError! ',e)
except BaseException  as e:
     print('类型错误 ValueError! ',e)
else:
    print('恭喜：没有任何异常')
print('hello world')
```

输出：

```
Enter x: 34
Enter y: 45
0.7555555555555555
Enter z: 3
11.333333333333334
恭喜：没有任何异常
hello world
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

7.2.5 finally 子句

和 else 子句不同，try 异常处理语句后面的 **finally** 子句无论有没有出现异常都会被执行。例如，执行以下代码：

```
try:
    x = 1/1
    print(x)
finally:
    print('无论是否出现异常，是否处理异常，总是会执行这一句')
```

执行上述代码的输出：

```
1.0
无论是否出现异常，是否处理异常，总是会执行这一句
```

上述代码的 try 中永远不会抛出异常，但 finally 子句还是会被执行。同样，即使 try 抛出异常，也会执行 finally 子句。例如：

```
try:
    x = 1/0
    print(x)
finally:
    print('无论是否出现异常，是否处理异常，总是会执行这一句')
```


执行上述代码的结果：

```
无论是否出现异常，是否处理异常，总是会执行这一句
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-5e8874c6e65a> in <module>()
      1 try:
----> 2     x = 1/0
      3     print(x)
      4 finally:
      5     print('无论是否出现异常，是否处理异常，总是会执行这一句')

ZeroDivisionError: division by zero
```

如果出现了异常，但没有处理，则当 finally 中的所有代码都执行完毕后，会继续向上一层的调用函数抛出这个异常。直到这个异常在某一层得到了处理，程序才会继续执行下去，或者一直没有处理这个异常，那么程序会终止执行。例如：

```
try:
    x = 1/0
    print(x)
except ZeroDivisionError as e:
    print('出现了除 0 的错误:ZeroDivisionError:',e)
finally:
    print('无论是否出现异常，是否处理异常，总是会执行这一句')
```

执行程序会抛出 ZeroDivisionError 异常并被 except 子句捕获处理：

```
出现了除 0 的错误:ZeroDivisionError: division by zero
无论是否出现异常，是否处理异常，总是会执行这一句
```

无论是否发生异常，finally 子句在程序离开 try 块后都一定会被执行。因此，finally 子句的作用主要是做一些清理工作，如释放外部资源(文件或网络连接等)，无论它们在使用过程中是否出错。

例如，在下面代码中能正常地打开文件但是在读取文件时出现了异常，当在 except 块中处理完异常后，打开的文件并没有被关闭：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
try:
    f = open('text.txt', 'r')
    line = f.readline()
    f.close()
except IOError:
    print('IOError')
```

为了能正确地关闭该文件，可以在最后添加一个 finally 块，以保证无论是否发生异常，都能关闭打开的文件。

```
f = None
try:
    f = open('text.txt', 'r')
    lines = f.readlines()
except IOError:
    print('IOError')
finally:
    f.close()
```

7.2.6 用 raise 抛出异常

Python 解释器在运行时可以自动抛出异常，程序员也可以在任何代码处用 raise 抛出异常。

例如，try 块可以用 raise 抛出一个异常：

```
try:
    score = float(input('请输入分数:'))
    if score <0:
        raise ValueError("出现了值异常错误！")
except ValueError as e:
    print(e)
```

程序员还可以在 except 块中捕获 try 块出现的异常，并进行处理，然后接着又抛出这个异常，这样做的目的是程序员可以首先自己处理一下异常，然后再交给上一层的调用者去继续处理这个异常。例如：

```
try:
    score = float(input('请输入分数:'))
    if score <0:
        raise ValueError("出现了值异常错误！")
except ValueError as e:
    print(e)
    raise e    #继续抛出异常对象 e,交给该代码的调用者去处理这个异常
```

7.2.7 自定义异常类

程序员也可以在自己的程序中定义新的异常类，自定义的异常类应该直接或间接地从 Exception 类派生。例如：

```
class MyError(Exception):
   def __init__(self, value):
      self.value = value
```

程序员可以用以下代码测试自己定义的异常类：

```
try:
    raise MyError("我自己定义的异常")
except MyError as e:
    print(e)
```

程序输出：

```
我自己定义的异常
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

7.2.8 预定义清理行为

用 finally 子句来保证 try 块中的资源得到释放的方法显得有点烦琐。而有些对象定义了标准的清理行为，该行为确保无论对象操作是否成功，当不再需要该对象的时候清理工作都会起作用。例如，下面的代码忘记关闭文件，可能会导致一些问题(如其他线程的程序可能无法打开该文件)：

```
f = open("text.txt")
for line in f.readlines():
    print(line)
```

此时，可以用 **with** 语句进行清理。with 语句可以确保文件之类的对象总能及时准确地被清理：

```
with open("text.txt") as f:
    for line in f:
        print(line)
```

7.3 调试程序

所谓调试程序就是找到引起程序错误的原因。

语法错误通常很容易调试，因为错误消息会显示出语法错误在程序中的第几行，所以很容易找到并修复它。

运行过程中的错误有时较难调试，这是因为错误消息和回溯可以准确地提示错误发生的位置，但并不一定会提示问题是什么。有时它们是由显而易见的问题引起的，如标识符名称不正确，但有时它们是由程序的特定状态触发的，但并不能明确指出众多可能的因素中哪一个不正常。

逻辑错误是最难修复的，因为它们不会导致任何可以追溯到代码中特定行的错误。如果只知道程序的运行行为或结果不正常，但不知道哪个位置出现了错误，那么有时跟踪导致错误行为的代码区域可能需要很长时间。例如，一名测试员会对某个功能用不同的输入进行测试，查看会产生哪些逻辑错误，然后通知程序员这些逻辑错误，程序员就要想办法去找出引起这些逻辑错误的原因。

专业术语“调试程序”是指找出程序中的逻辑错误这个过程。

调试一个程序通常有下面几种方法。

- 用函数 print()等在代码的不同位置输出一些数据，查看这些输出值是否和预期值一致。
- 在代码中不同位置插入断言语句，检查某个条件是否满足。
- 使用日志 logging 功能，类似 print()的调试方法，只不过日志是将调试信息输出到文件中，以方便程序员通过日志文件分析程序的错误原因。
- 用调试工具通过设置断点或单步执行检查相应变量值，找出引起错误的原因。

7.3.1 输出(打印)

快速简单地调试程序的方法是利用输出语句输出变量的值或中间计算结果。例如，在函数中用函数 print()输出运算过程的中间结果，帮助判断这个函数的每一步计算是否正确：

```
import math
def length(x, y):
    print("x is {} and y is {}".format (x, y))  #检查输入的变量是否正常
    x_2 = x**2
    print("x^2=",x_2)                           #检查 x 的平方是否正确
    y_2 = y**2
    print("y^2=",y_2)                           #检查 y 的平方是否正确
    z_2 = x_2 + y_2
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        print("z^2=",z_2)                                    #检查 x 的平方与 y 的平方的和是否正确
        z = math.sqrt(z_2)
        print("z",z)                                         #检查 z 的平方根是否正确
        return z

    length(-2,3)
```

输出：

```
x is -2.000000 and y is 3.000000
x^2= 4
y^2= 9
z^2= 13
z 3.605551275463989
3.605551275463989
```

采用这种打印的方式有如下缺点。

- 一旦函数能正确工作，程序员可能会删除所有打印语句，因为他们不希望程序一直打印这些调试信息。但是，代码经常发生变化，下次再次测试这个函数时，又必须重新添加打印语句。
- 为了下次再次使用这些打印语句，可能不是删除它们而是注释它们。但代码改变时，打印语句也要跟着变化，最终会使这些适应不同变化的打印语句越来越多、越来越乱。
- 为了打印中间结果，需要人为地将一步计算变成几步来计算，使得代码变得复杂和冗长。而实际上，最后真正的代码要简洁很多：

```
def length(x, y):
    return math.sqrt(x**2 + y**2)
```


7.3.2 断言

另一种调试程序的方法是使用**断言**(**assert**)来检查某个条件或结果。例如：

```
assert(3**2 == 3*3)
assert (1+1==3)
```

执行第二条语句是因为条件不满足而抛出“AssertionError”异常：

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-7-424a15871c20> in <module>()
     1 assert(3**2 == 3*3)
----> 2 assert (1+1==3)

AssertionError:
```

断言(**assert**)用于判断一个表达式的值是否为 True，如果断言失败，则 assert 语句本身就会抛出 AssertionError。还可以在断言的后面用逗号,跟一个表达式，如果断言失败，则输出这个表达式的值。

```
assert (1+1==2),'1+1=2 不成立'
assert (1+1==3),'1+1=3 不成立'
```

执行第二条语句时抛出异常：

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-4-21459b42123a> in <module>()
     1 assert (1+1==2),'1+1=2 不成立'
----> 2 assert (1+1==3),'1+1=3 不成立'
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
AssertionError: 1+1=3 不成立
```

因此，assert 的使用格式为：

```
assert (condition)
```

或

```
assert (condition),(error_message)
```

对于第二种形式，如果条件表达式 condition 的值为 False，则会在断言的 AssertionError 异常后面输出 error_message 的内容。

断言(assert)相对于输出语句有一个好处就是不需要删除断言语句，只要在优化模式下执行程序就会自动关闭调试模式。此时，不会执行断言检查和输出调试信息。

Python 解释器执行一个程序时，设置–o 选项可关闭调试模式，以优化模式执行程序。例如：

```
python -o my.py
```

7.3.3 日志

和 print()类似，logging(日志)也是一种可用于调试程序的方法。通过在程序的不同代码行添加 logging 语句，可以将程序的信息记录到文件中，而不是直接输出到控制台窗口。然后通过分析这个日志文件，可以检查程序的运行情况，查看程序是否出现了错误。日志文件可以记录大量日志信息，而 print()在控制台窗口的信息不便查看。和 assert 相比，logging 不会抛出错误。

使用 Python 的 logging 记录日志几乎与 print()语句类似，在记录某些信息地方插入 logging 语句即可。logging 提供不同的方法用来记录不同级别的信息。logging 信息的级别按顺序从最高值(最严重)到最低值(最不严重)分别是：

- CRITICAL(关键)——非常严重的错误；
- ERROR(错误)——不太严重的错误；
- WARNING(警告)——警告；
- INFO(信息)——一些信息；
- DEBUG(调试)——调试消息。

针对这些不同级别的信息，分别有不同的方法，如表 7-1 所示。

表 7-1 logging 的成员函数及说明

函 数	说 明
logging.debug(msg, ```*args, **kwargs```)	创建一条严重级别为 DEBUG 的日志记录
logging.info(msg, ```*args, **kwargs```)	创建一条严重级别为 INFO 的日志记录
logging.warning(msg, *args, **kwargs)	创建一条严重级别为 WARNING 的日志记录
logging.error(msg, *args, **kwargs)	创建一条严重级别为 ERROR 的日志记录
logging.critical(msg, *args, **kwargs)	创建一条严重级别为 CRITICAL 的日志记录
logging.log(level, *args, **kwargs)	创建一条严重级别为 level 的日志记录
logging.basicConfig(**kwargs)	对 root logger 进行一次性配置

其中，函数 logging.basicConfig(**kwargs)用于指定“要记录的日志级别”“日志格式”“日志输出位置”“日志文件的打开模式”等信息，其他几个函数都是用于记录各个级别日志的函数。

通常，首先用 basicConfig 配置记录的级别如 logging.DEBUG，表示是 DEBUG 以上级别的信息才会被记录到日志里。例如：

```
import logging
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
logging.basicConfig(filename= 'my.log',level=logging.ERROR)
logging.debug("This is a debug log.")
logging.info("This is a info log.")
logging.warning("This is a warning log.")
logging.error("This is a error log.")
logging.critical("This is a critical log.")
```

上面的代码将 ERROR 以上级别的日志记录到日志文件 my.log 中，my.log 文件内容如图 7-1 所示。

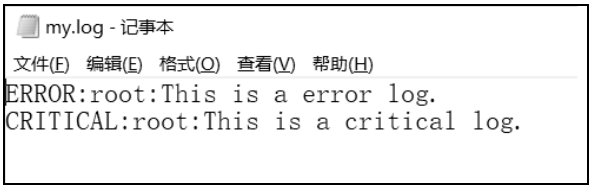


图 7-1　my.log 文件内容

除用不同的函数如 debug()、info()等外，也可以只用一个函数 log()，但设置信息的级别作为 log()方法的第一个参数。例如：

```
import logging

logging.basicConfig(filename= 'my.log',level=logging.ERROR)
logging.log(logging.DEBUG, "This is a debug log.")
logging.log(logging.INFO, "This is a info log.")
logging.log(logging.WARNING, "This is a warning log.")
logging.log(logging.ERROR, "This is a error log.")
logging.log(logging.CRITICAL, "This is a critical log.")
```


7.3.4　调试工具

有一些自动化工具可以帮助调试错误，并保持代码尽可能正确，从而最大限度地减少新错误蔓延的可能性。其中一些工具用于分析程序的语法，可报告错误和糟糕的编程风格，而另一些的工具则分析正在运行的程序。常见的第三方调试工具包括 Pyflakes、pylint、PyChecker、pep8 等。另外，集成开发环境如 PyCharm 也提供了相应的调试工具。

1．pdb 模块

Python 的 pdb 模块是 Python 内置的一个调试工具。借助 pdb 模块，程序员可以设置断点、逐行执行程序(单步调试)、进入函数调试、查看变量的值、动态改变变量的值、查看函数调用栈，并在程序崩溃时执行程序的“事后验证”。

使用 pdb 模块有以下两种方式。

- 在运行代码文件时将其作为脚本调用。例如：

```
python -m pdb my_script
```

这会触发调试器在脚本第一行指令处停止执行。

- 在代码中导入该模块并在代码中使用它的内置函数。例如：

```
import pdb                      #导入模块 pdb

def fun():
    pdb.set_trace()             #设置断点
    bad_idea = 3 + "4"

pdb.run('fun()')                #调试执行函数 fun()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

在代码中通过“pdb.set_trace()”设置断点，然后运行这个脚本文件，程序执行到断点处就会暂停。

2．pdb 模块常用调试命令

无论使用哪种调试模式，都可以使用 pdb 模块提供的调试命令调试程序。例如，用各种特殊的字母代表的命令调试程序：

w：显示当前的上下文信息；
n：单步执行每条语句；
s：进入函数内部执行；
a：打印当前函数的参数列表；
c：继续执行直到下一个断点。
更多命令可查看官方文档或在网络上搜索。
建议用更专业的集成开发环境，如 PyCharm 调试 python 程序。

总结

- 程序的错误分为语法错误和运行时错误，而运行时错误又分为异常错误和逻辑错误。
- 可以用 try...except 程序块对不同的异常错误进行异常处理。异常没有发生时，会执行 else 子句，而无论异常是否发生或是否处理异常，都会执行 finally 子句。
- 程序员可以自己定义异常类型，也可由程序员自己抛出各种异常。
- 自定义清理语句，如 with 语句，可以确保在任何情况下资源都能得到释放。
- 调试程序的方法主要有打印(print())、断言(assert)、日志记录(logging)、用调试工具(pdb 模块)等设置断点或单步调试。

7.4 习题

1．找出下列代码中的语法错误，并解释错误的原因。

```
(1)myfunction(x, y):
    return x + y
(2)else:
    print("Hello!")
(3)if mark >= 50
    print("You passed!")
(4)if arriving:
     print("Hi!")
  esle:
     print("Bye!")
(5)if flag:
  print("Flag is set!")
```

2．找出下列代码中的运行时错误(逻辑错误)，并解释错误的原因。

```
(1)product = 0
  for i in range(10):
     product *= i
(2)sum_squares = 0
  for i in range(10):
     i_sq = i**2
  sum_squares += i_sq
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
(3)nums = 0
   for num in range(10):
      num += num
```

3．一个 try…except 程序块中可以有多少 except 子句？（　　）

A．0　　B．1　　C．多于 1 条　　D．多于 0 条

4．打开一个不存在的文件会发生什么？（　　）

A．创建一个新文件　　B．什么也不发生

C．抛出一个 exception　　D．上述说法都不对

5．try…except…else 的 else 子句在什么情况下被执行？（　　）

A．总会执行　　B．异常发生时才执行

C．没有异常发生时才执行　　D．当 except 块中发生异常时才执行

6．下列代码的输出结果是什么？（　　）

```
def foo():
    try:
        return 1
    finally:
        return 2
k = foo()
print(k)
```


A．1　　B．2　　C．3

D．错，在一个 try…finally 有多于一个 return 语句

7．下列代码的输出结果是什么？（　　）

```
number = 5.0
try:
    r = 10/number
    print(r)
except:
    print("Oops! Error occurred.")
```

A．Oops! Error occurred.　　B．2.0

C．2.0 Oops! Error occurred.　　D．None object

8．下列代码的输出结果是什么？（　　）

```
try:
  #code that can raise error
   pass

except (TypeError, ZeroDivisionError):
    print("TWO")
```

A．只要出现异常 exception（无论什么异常）就打印 TWO

B．不出现异常 exception 时就打印 TWO

C．出现 TypeError 或 ZeroDivisionError 异常就打印 TWO

D．当 TypeError 和 ZeroDivisionError 异常都出现时才打印 TWO

9．对字符串 strtext = “Welcome”，下列哪条语句会产生 TypeError？（　　）

A．strtext[1]= ‘r’　　B．print（strtext[0]）　　C．print（strtext.strip()）

D．print（strtext.lower()）　　E．None of these

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

10．下列代码的输出结果是什么？（　　）

```
def  Test():
    try:
        print(20)
    finally:
        print(30)
Test()
```

A．30
20

B．20
30

C．30

D．20
5

E．None

11．下列代码的输出结果是什么？（　　）

```
try:
    print("throw")
except:
    print("except")
finally:
    print("finally")
```

A．finally
except

B．throw
finally

C．finally
throw

D．Syntax error

E．except
finally

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 8 章　高级语法特性

8.1　容器、可迭代对象、迭代器、生成器

8.1.1　容器

容器是存储多个值(对象)的数据类型，如 list，tuple，set，dict 等都是容器。

由于这些容器类都实现了一个叫作__contains__()的方法，因此可以使用 in 或 not in 来判断某个值(对象)是否存在于某个容器内。例如：

```
print (1 in [1, 2, 3])                  #True
print (2 not in (1, 2, 3) )             #False
print ('a' in ('a', 'b', 'c') )        #True
print ('x' in 'xyz' )                   #True
print ('a' not in 'xyz')                #True
```

如果一个类(如 A)实现了__contains__方法：

```
class A(object):
    def __init__(self):
        self.items = [1, 2]
    def __contains__(self, item):   #item 是否在 A 的对象中
        return item in self.items
```

那么就可以用 in 或 not in 来判断某个值(对象)是否存在于这个类(A)的对象中：

```
a = A()
print (1 in a )   #True
print (2 in a )  #True
print (3 in a )   #False
```

输出：

```
True
True
False
```

8.1.2　可迭代的和迭代器

1. 迭代

“**迭代**”是指逐个地遍历容器对象的每个元素。例如，对于 str、list、tuple 等容器类的对象：

```
s= 'hello world'
for e in s:                     #迭代访问字符串 s 中的每个字符
    print(e,end = ',')
print()
atuple = (3,5,7,9)
for e in atuple:                #迭代访问 tuple 对象 atuple 中的每个字符
    print(e,end = ',')
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
h,e,l,l,o, ,w,o,r,l,d,
3,5,7,9,
```

然而对于容器类 A 的每个对象 a，虽然 a 是一个包含两个元素“1”和“2”的容器对象，但不是“可迭代的”：

```
a = A()
for e in a:
   print(e,end = ',')
```

执行会抛出 TypeError 异常：

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-27-3e181109f1a0> in <module>()
     1 a = A()
----> 2 for e in a:
     3     print(e,end = ',')
TypeError: 'A' object is not iterable
```

当试图迭代这个对象 a 时，抛出了 TypeError 类型的异常错误：“'A' object is not iterable”（A 的对象不是“**可迭代的**”**(iterable)**）。而内在容器类型，如 str、list、tuple 等的对象都是可迭代的(iterable)。

可以用 isinstance()函数来测试对象是否为可迭代的(iterable)。例如：

```
from collections import iterable
print( isinstance(a, iterable) )                #A 类的对象 a 是否可迭代的
print( isinstance('abc', iterable)  )           #str 是否为可迭代的
print( isinstance([1,2,3], iterable) )          #list 是否为可迭代的
print( isinstance(123, iterable) )              #整数是否为可迭代的
```

输出：

```
False
True
True
False
```

2．可迭代的和迭代器的区别

一个对象是**可迭代的**(iterable) 是指它所属的类定义了一个可以返回一个**迭代器(iterator)**的**__iter__()**方法，该对象称为可迭代对象。

一个对象是一个**迭代器**(iterator) 是指它所属的类定义了**__next__()**方法，该方法要么返回可迭代对象的下一个元素，要么抛出(raise)一个 StopIteration 异常。

简单地讲就是，一个可迭代的对象的__iter__()方法返回一个迭代器，而这个迭代器的__next__()方法会返回可迭代对象的下一个元素。

Python 提供了两个对应的内置函数 **iter()**和 **next()**，内置函数 iter()作用于可迭代对象，会调用这个对象的**__iter__()**方法返回一个迭代器。而内置函数 next()作用于这个迭代器，会调用这个迭代器的**__next__()**方法返回可迭代对象的下一个元素。

例如，下面代码中的 list 对象 alist 就是一个可迭代对象，通过 iter(alist)方法就能够返回一个迭代器 it，next(it)方法返回可迭代对象 alist 的下一个元素：

```
alist = [3,5,19,7]
it= iter(alist)                 #返回一个迭代器
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
next(it)                                    #得到迭代器指向的下一个值
```

输出：

```
3
```

通过不断调用 next() 函数，可以遍历迭代器指向的可迭代对象 alist 中的每一个元素。例如：

```
next(it)
```

输出：

```
5
```

借助 for 循环就可以迭代访问一个可迭代对象的所有元素：

```
for e in alist:
   print(e,end = ',')
```

输出：

```
3,5,19,7,
```

综上所述，可迭代对象返回迭代器，迭代器的_next_() 方法返回可迭代对象的下一个元素，迭代过程如图 8-1 所示。

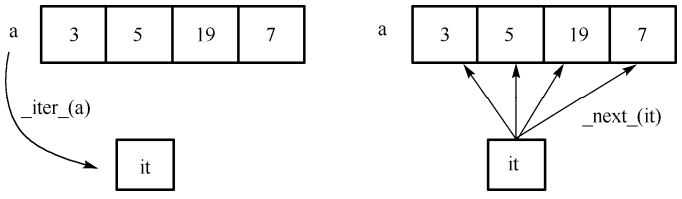


图 8-1　迭代过程

对于熟悉 C 语言的读者，可以将可迭代对象看作一个数组，数组名就是指向数组中原元素的指针，这个指针可以看作从这个数组返回的一个迭代器，然后通过*运算符作用于这个指针，就得到了数组的一个元素，这个指针（迭代器）可以递增指向下一个（next）数组元素，通过这个指针迭代指向数组的每一个元素，从而遍历数组的所有元素。

迭代器是用于记录可迭代对象的当前元素位置的一个对象，迭代器可以通过__next__() 方法的移动指向下一个元素。

例如，下列的类 A 实现了__iter__() 方法，因此类 A 的对象就是一个可迭代对象。__iter__() 方法必须返回一个迭代器，即类 B 的对象必须是一个迭代器，也就是说，类 B 必须实现__next__() 方法。

```
class A(object):
   def __iter__(self):
      return B()
```

假如定义一个如下的类 B：

```
class B(object):
    def __next__(self):
      return "hello"
```

下面是一个例子：

```
iterable = A()                  #iterable 是一个可迭代对象
iterator = iter(iterable)       #返回一个迭代器
next(iterator)
```

输出：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
'hello'
```

可以这样用函数 next()不断访问可迭代对象的下一个元素。

```
next(iterator)
```

输出：

```
'hello'
```

可利用 for 循环基于函数 iter()和函数 next()来迭代访问一个可迭代对象的每个值：

```
for e in iterable:
    print(e)
```

但这个循环过程会一直循环下去，因为没有通过关键字 raise 抛出 StopIteration 异常。输出如下：

```
hello
hello
hello
hello
hello
...
```

可以修改上面的__next__()方法以抛出 StopIteration 异常，表示所有元素都已经访问完：

```
class B(object):
    def __init__(self):
        self.i = 0
        self.data = [2,4,8]
    def __next__(self):
        if self.i==3 :
            raise StopIteration()
        self.i+=1
        return self.data[self.i-1]
iterable = A()
for e in iterable:
    print(e)
```

输出：

```
2
4
8
```

通常，可迭代对象的所属类和迭代器的所属类并不是这样完全分离的，而是同一个类，这个类同时实现__iter__()方法和__next__()方法。因此，这个类的对象既是一个可迭代对象也是一个迭代器。例如：

```
class X(object):
    """这个类的对象既是可迭代的对象也是一个迭代器，因为实现了__iter__和 next 方法"""
    def __init__(self):
        self.i = 0
        self.data = [2,4,8]

    def __iter__(self):
        return self                    #返回的迭代器就是可迭代对象自身

    def __next__(self):
        if self.i==3 :
            raise StopIteration()
        self.i+=1
        return self.data[self.i-1]
```

因此，X 类的对象即是一个可迭代对象也是一个迭代器，如果作为可迭代对象，则记录了多个

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

数据元素；如果作为迭代器，则记录了当前访问元素的位置。因此，既可以用函数 iter()和函数 next()，也可以用 for 循环访问其中的元素。例如：

```
b = X()
it = iter(b)          #得到迭代器
print(next(it))       #返回迭代器指向的当前元素，迭代器指向下一个元素
print(next(it))

b2 = X()
for e in b2:
  print(e,end = ' ')
```

输出：

```
2
4
2 4 8
```

3. __getitem__()方法

一个对象是可迭代的，除其所属类实现__iter__()方法外，其所属类还可实现__getitem__()方法。例如：


```
class Y:
  def __getitem__(self, i):
    return i*i
```

这个 Y 类的对象也是可迭代对象。例如：

```
y = Y()
for e in y:
  print(e, end=' ')
```

输出将无限循环：

```
0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400 441 484
529 576 625 676 729 784 841 900 961 1024 1089 1156 1225 1296 1369 1444 1521 1600
1681 1764 ...
```

__getitem__()方法的格式是：

```
__getitem__(self,key)
```

即其第二个参数实际上是一个键(关键字)，对实现该方法的类对象 obj，可以通过 obj[key]来得到该键对应的值，即该类实际上类似 Python 的 dict 类。例如：

```
print(y[2])
print(y[4])
```

输出：

```
4
16
```

和__getitem__()方法对应的还有一个__setitem__()方法，用于设置一个键的值，其格式是：

```
__setitem__(self,key,value)
```

下面是一个实现了这两个方法的类：

```
class MyDict:
   def __init__(self):
      self.dict = {}
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    def __getitem__(self,key):
        return self.dict[key]
    def __setitem__(self,key,value):
        self.dict[key] = value
    def __len__(self):
        return len(self.dict)

d = MyDict()
d['李平']=67.5          #调用__setitem__()方法
print(d['李平'])        #调用__getitem__()方法,输出 67.5
d['王伟'] = 89.6
print(d['王伟'])        #调用__getitem__()方法,输出 89.6
```

输出：

```
67.5
89.6
```

8.1.3 生成器

生成器既是一个用于创建迭代器的简单而强大的工具，又是一个可以产生一系列值的函数。生成器会记住上次执行的语句。每当需要数据时，生成器都会从中断处继续，并通过 yield 语句产生一个新的数据。因此，生成器是一个特殊的迭代器，也是一个可迭代对象。

可以用两种方式创建生成器：

- 生成器函数；
- 生成器表达式。

1. 生成器函数

生成器函数是指包含 yield 关键字的函数，其格式是：

```
#生成器函数
def gen(n):
    for i in range(n):
        yield i
```

函数 gen()就是一个生成器函数，其中以关键字 yield 开头的语句会产生一个返回值。通过传递相应的参数给这个生成器函数，可以创建一个生成器对象。例如：

```
g = gen(5)                  #创建一个生成器
print (g)
print ( type(g) )           #<type 'generator'>
```

输出：

```
<generator object gen at 0x0000025B76E52BF8>
<class 'generator'>
```

g 是一个 generator object(生成器对象)，它的类型是 generator(生成器)，可以通过这个生成器对象调用__next__()方法，生成器函数会执行产生一个数据。例如：

```
g.__next__()
```

输出：

```
0
```

生成器会记住函数执行的状态，当再次调用__next__()方法时，会根据上次的位置和状态(i=0)继续执行，当再遇到 yield 语句时，又产生一个新的返回值并终止当前的执行。例如：

```
g.__next__()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
1
```

可以看到，每次调用生成器的__next__()方法，都会接着上次中断的位置继续执行，并产生一个新的数据。当然，也可以调用内置函数作用于生成器对象(该函数内部仍然调用的是生成器对象的__next__()方法)。例如：

```
next(g)
```

输出：

```
2
```

执行：

```
next(g)
```

输出：

```
3
```

既然生成器是一个可迭代对象，那么当然可以用 for 循环迭代其产生的序列。例如：

```
#迭代
g = gen(10)
for i in g:
   print (i,end = " ")
```


输出：

```
0 1 2 3 4 5 6 7 8 9
```

生成器函数与包含 return 的函数执行机制区别如下。

- 包含 return 的函数当遇到 return 语句时，会停止函数执行并返回，函数的每次执行都是独立的，且是从头开始执行的，不会保存上一次执行的状态和位置。
- 包含 yield 的生成器函数用于产生一系列值，每次执行时当遇到 yield 语句时会停止执行并返回 yield 语句的结果，但内部会保留上一次执行的状态，下一次执行时将从上一次的 yield 之后的代码继续执行，直到再次遇到 yield 时返回。

生成器函数可以用来产生一系列的值，但这些值并没有同时保存在内存中，而是在需要时才产生一个值，因此，如果要产生的值的集合很大，则可以节省很多内存，这是因为普通方法产生的所有数据都要同时在存储内存中，这会导致很大的空间消耗。例如：

```
#求 1 到 10 之间的整数和
alist = [0,1,2,3,4,5,6,7,8,9,10]
sum = 0
for e in alist:
   sum += e
print(sum)
```

输出：

```
55
```

上述代码计算“0”到“10”之间的整数和，需要首先将这些整数保存在一个 list 对象中，10 个整数对象都需要内存，这个 list 对象中的 10 个整数的引用也需要内存。但是，如果用生成器函数，则每次只要保存一个整数的内存就可以了：

```
g = gen(11)
sum = 0
for e in g:
   sum += e
print(sum)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
55
```

Python 的 range()函数就是一个生成器函数，和上例的 gen()函数一样产生某一个范围里的值。上述代码可以改写成调用 range()函数：

```
sum = 0
for e in range(11):
    sum += e
print(sum)
```

输出：

```
55
```

可以用生成器函数产生斐波拉契数列。例如：

```
def fib(M):
    n, a, b = 0, 0, 1
    while n < M:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```


用生成器函数可以产生一个具体的生成器对象，如 fib(8)。例如，通过 for 循环输出该对象中的所有元素：

```
for x in fib(8):
    print(x,end = " ")
```

输出：

```
1 1 2 3 5 8 13 21
```

2. 生成器表达式

生成器表达式是一种更简单的创建生成器对象的方式。它的形式类似列表解析式，但将列表解析式的[]换为()。例如：

```
g = (i for i in range(5)) #创建一个生成器

print(g)
it = iter(g)              #生成器是一个可迭代的对象，函数 iter()可作用于它，是返回的是它自身
print(it)

print(next(g))            #生成器是一个迭代器，函数 next()可作用于它，以产生下一个元素
print(next(g))
```

输出：

```
<generator object <genexpr> at 0x000001D740540780>
<generator object <genexpr> at 0x000001D740540780>
0
1
```

当然，可以用 for 循环迭代访问生成器对象的所有元素。例如：

```
#继续用 for 循环迭代访问其元素
for i in g:                    #生成器也是一个可迭代对象
  print (i,end = " ")
```

输出：

```
2 3 4
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

8.1.4 例子：读取多个文件

假如要编写一个程序，首先输入一个包含多个文件名的 list 对象，然后输出每个文件的内容，则可以这样编写：

```
def cat(filenames):
    for f in filenames:
        for line in open(f):
            print(line)
```

假如要输出的是包含某种特殊子串的行，则可能又要编写另外一个函数：

```
def grep(pattern, filenames):
    for f in filenames:
        for line in open(f):
            if pattern in line:
                print(line)
```

两段程序都有许多公共的代码，但很难将这些公共代码编写到一个函数中去，因为这两段程序都是直接对每个行进行处理的。如果用生成器来实现以上功能，就可以将“读取每个文件的每一行”的公共代码编写到生成器中，从而简化代码，提高代码的复用性。例如：


```
def readfiles(filenames):
    for f in filenames:
        for line in open(f):
            yield line
```

这个生成器函数 readfiles()可以每次读取一个文件的一行，借助该函数，可以重新编写上述的两段代码：

```
#读取并显示所有文件的内容
def printlines(lines):
    for line in lines:
        print line,
def cat(filenames):
    lines = readfiles(filenames)
    printlines(lines)

#读取并显示所有文件特定模式的行
def grep(pattern, lines):
    return (line for line in lines if pattern in line)



def grep(pattern, filenames):
    lines = readfiles(filenames)
    lines = grep(pattern, lines)
    printlines(lines)
```

将读取多个文件的每一行的代码放到一个生成器函数 readfiles()中，该函数返回一个代表所有文件所有行的迭代器对象 lines。利用函数 printlines()迭代访问 lines 的每一行并输出。

函数 grep(pattern, lines)接收迭代器对象 lines，并返回一个生成器表达式，通过这个生成器表达式返回迭代器对象 lines，函数 printlines()迭代访问 lines 的每一行并输出。

因为函数 readfiles()和 grep 函数(pattern, lines)返回的都是迭代器对象，避免了将所有文件的所有行一次读进内存，而且在具体 for 循环访问时每次产生和处理一行，所以只要存储文件的一行，就可以处理任意多、任意大的文件。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

8.1.5 标准库的迭代器工具

1. 函数 zip()

函数 zip()的格式如下：

```
zip(*iterables)
```

该函数的输入参数是数目可变的任意多个可迭代对象，返回一个数据元素(tuple 的迭代器)，每个 tuple 是由多个输入迭代器中对应的元素构成的。

函数 zip()的返回值。

- 如果没有输入参数，则返回一个空的迭代器.
- 如果输入一个可迭代对象，则返回一个数据元素(tuple 的迭代器)，每个 tuple 只有一个值。
- 如果输入多个可迭代对象，则返回一个数据元素(tuple 的迭代器)，每个 tuple 由分别来自输入的可迭代对象的对象元素构成。

例如：

```
result = zip()                          #result 是一个空的迭代器
print(list(result))

alist = [1,2,3]
result = zip(alist)                     #result 是一个数据元素(tuple 的迭代器)
for e in result:                        #每个 e 是一个对象构成的 tuple
    print(e,end = ' ')
print()

str_tuple = ('he','world','abc')
result = zip(alist,str_tuple)           #result 是一个数据元素(tuple 的迭代器)
for e in result:                        #每个 e 是两个对象构成的 tuple
    print(e,end = ' ' )
```

输出：

```
[]
(1,) (2,) (3,)
(1, 'he') (2, 'world') (3, 'abc')
```

可以通过 for 循环迭代访问一个 zip 对象中的每个元素。例如：

```
for x, y in zip(["a", "b", "c"], [1, 2, 3,4 ]):
  print(x, y)
```

输出：

```
a 1
b 2
c 3
```

zip 对象是一个可迭代对象，因此可以用它构造一个 list 对象，但是不能用它构造一个 tuple 对象。因为可迭代对象每次产生一个值，而 tuple 对象是不可修改的，无法每次向 tuple 对象中添加迭代过程中产生的新元素。例如：

```
t2 = zip(["a", "b", "c"], [1, 2, 3,4 ])
alist = list(t2)
print(alist)
atuple = tuple(t2)
print(atuple)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
[('a', 1), ('b', 2), ('c', 3)]
()
```

再看一个例子：

```
a = ['a', 'b', 'c']
b = [3, 4, 5, 8]

c = zip(a, b)                    #c 是一个迭代器对象
clist = list(c)                  #从 c 构造一个 list 对象
print(clist)

d, e =  zip(*clist)              #用*和 zip 结合进行解包
print('d =', d)
print('e =', e)
```

clist 作为一个元素是 tuple 对象的 list 对象，用*对其“解封参数列表”(*clist)并传递给函数 zip()，函数 zip()接收了三个可迭代对象，('a', 3)、('b', 4)和 ('c', 5)。从而产生了有两个 tuple(('a', 'b', 'c')和(3, 4, 5))的可迭代对象。

输出：


```
[('a', 3), ('b', 4), ('c', 5)]
d = ('a', 'b', 'c')
e = (3, 4, 5)
```

2．函数 enumerate()

输入一个可迭代的对象，返回一个 enumerate 对象，该对象也是一个可迭代对象，其每个元素形如(index,value)。其中，value 来自输入的可迭代对象。例如：

```
for i, c in enumerate(["a", "b", "c"]):
   print(i, c)
```

输出：

```
0 a
1 b
2 c
```

函数 enumerate()实际上还有第 2 个参数，用于指定下标的起始值。例如：

```
for i,c, in enumerate(["a", "b", "c"], 4);
    print(i, c)
```

输出：

```
4  a
5  b
6  c
```

3．Itertools 模块

标准库的 itertools 模块提供了很多和迭代器相关的工具。

(1) **函数 chain()**返回将多个可迭代对象串联(chains)起来的可迭代对象。例如：

```
import itertools

it1 = iter([1, 2, 3])
it2 = iter([4, 5, 6])
it= itertools.chain(it1, it2)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    for e in it:
        print(e, end = " ")
```

输出：

```
1 2 3 4 5 6
```

(2) 函数 count() 格式如下：

```
count(start=0, step=1)
```

该函数返回一个迭代器，迭代器产生一个均匀的数字序列，该序列的开始值是 start，两个相邻值的差是 step，这个序列是一个无限序列。例如：

```
from itertools import count
for i in count(10):
    if i > 20:
        break
    else:
        print(i,end = " ")
```

输出：

```
10 11 12 13 14 15 16 17 18 19 20
```

(3) 函数 islice() 也是 itertools 的一个函数，其函数格式如下：

```
islice(iterable, stop)
islice(iterable, start, stop[, step])
```

该函数用于从可迭代对象中产生一个切片对象(islice object)。其中，stop 是切片的结束位置，start 是开始位置，而 step 是步长。例如，可以截取 count 产生序列的前 5 个数字：

```
from itertools import islice
for i in islice(count(10), 5):
    print(i,end = " ")
```

输出：

```
10 11 12 13 14
```

(4) 函数 cycle()。

函数 cycle() 作用于一个可迭代对象，产生一个重复循环 iterable 对象中的值的迭代器。例如：

```
from itertools import cycle
count = 0
for item in cycle('XYZ'):
    if count > 9:
      break
    print(item,end=" ")
    count += 1
```

输出：

```
X Y Z X Y Z X Y Z X
```

(5) 函数 repeat(object[, times])。

该函数用于产生一个 times 个重复值为 object 的迭代器。它和函数 cycle() 的区别是，它只作用于只有一个值的对象 object，而函数 cycle() 作用于一个有多个值的可迭代对象。

```
from itertools import repeat
r = repeat(7, 3)                    #r 是一个产生 3 个 7 的可迭代对象
print(next(r),end = ' ')
print(next(r),end = ' ')
print(next(r),end = ' ')
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
7 7 7
```

当可迭代对象超过 3 个时，则会出错：

```
print(next(r))
```

抛出 StopIteration 异常：

```
---------------------------------------------------------------------------

StopIteration                             Traceback (most recent call last)

<ipython-input-43-0f1e116d63f7> in <module>()
----> 1 print(next(r))
StopIteration:
```

总结

本节介绍的容器(container)、可迭代对象(iterable)、迭代器(iterator)、生成器(generator)、生成器函数/生成器表达式之间的关系可以用图 8-2 表示。

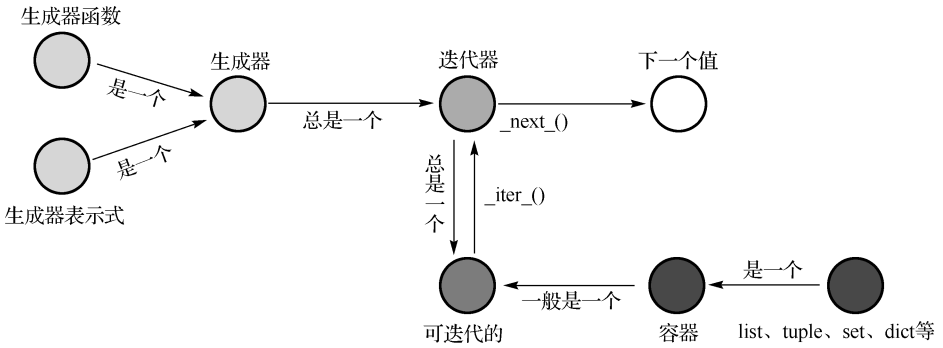


图 8-2　生成器函数/生成器表达式、生成器、可迭代对象、迭代器、容器之间的关系

- list、set、tuple、dict 都是容器，容器实现了__contains__()方法，可以通过 in 或 not in 判断一个值是否在一个容器内。
- 可迭代对象的所属类实现__iter__()方法，该方法返回一个迭代器。
- 迭代器是实现__next__() 方法的类的对象。__next__()方法要么返回可迭代的下一个元素，要么抛出(raise)一个 StopIteration 异常。可迭代对象所属的类通常同时实现这两个方法，因此该类对象既是可迭代对象也是迭代器对象。
- 生成器是一种特殊的迭代器，定义生成器有两种方法，生成器函数和生成器表达式。
- Python 有一些内置函数，如 zip()、enumerate()，可用于处理迭代器对象，标准库的 itertools 模块也有许多函数，如 chain()、count()、islice()、cycle()、repeat()等，可用于处理可迭代对象。

8.2　闭包

在理解闭包(Closure)概念之前，应首先理解**嵌套**函数(也称**局部函数**)和**作用域**。

8.2.1　作用域

作用域是指程序变量(或函数)的作用范围，即这些变量(或函数)能够被访问的范围。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

例如，一个函数内部定义的变量称为这个函数的**局部变量**，局部变量的作用范围只能在这个函数内部范围内，不能在函数外访问它。例如：

```
def fun(var):
   var = 5

print(var)    #在函数 fun()外部不能访问函数 fun()内部的局部变量 var
```

执行将抛出 NameError 异常：

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-3-926d6ec7b930> in <module>()
     2    var = 5
     3
----> 4 print(var)    #在 fun 函数外部不能访问 fun 函数内部的局部变量 var
NameError: name 'var' is not defined
```

出现“NameError: name 'var' is not defined()”即“名字 var 未定义”的语法错误，这是因为在函数 fun()外部不能访问函数 fun()内部的局部变量 var。

在函数外层定义的变量是全局变量，它是全局范围内可见的。当然，在任何函数里也可以访问它。例如：

```
var = 5
def fun():
   print(var)

fun()
print(var)
```

输出：

```
5
5
```

8.2.2 嵌套函数

在其他函数内定义的函数称为**嵌套函数(nested function)**或局部函数(local function)。例如：

```
def print_msg():
   msg = "hello"

   def printer():
      print('I am a local function')
      print(msg)

   printer()

print_msg()
```

输出：

```
I am a local function
hello
```

这个程序的执行语句是一个函数调用语句“print_msg()”，该函数内只有一条执行语句“printer()”，也是一条函数调用语句，但 printer()是函数 print_msg()内的一个局部函数。而这个局部函数 printer()第一句打印的是一个普通字符串，第二句打印的是其外部包围函数里的局部变量 msg。也就是说，嵌套函数里的代码是可以访问包围它的函数 print_msg()的局部变量的，正如一个函数可以访问全局变量一样。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

但是，如果在全局作用域调用函数 print_msg()内的局部函数 printer()，则会出错，正如不可以在全局作用域访问一个局部作用域的变量一样。例如：

```
printer()
```

执行这句代码后抛出异常：

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-40-df218a64109d> in <module>()
----> 1 printer()
NameError: name 'printer' is not defined
```

8.2.3 什么是闭包

Python 中的函数也是对象，因此完全可以在一个函数中返回另外一个函数对象(如嵌套函数)。例如：

```
def print_msg():
    msg = "hello"
    def printer():
        print('I am a local function')
        print(msg)

    return printer

p = print_msg()     #p 引用 print_msg()返回结果，即函数对象 printer()
```


上述代码执行函数 print_msg() 并将返回结果(一个嵌套函数对象 printer())赋值给变量 p。通过这个变量 p，可以执行它引用的函数 printer()：

```
p()
p()
```

输出：

```
I am a local function
hello
I am a local function
hello
```

p 引用的嵌套函数 printer()可以访问其包围函数 print_msg()的局部变量，而一个函数的局部变量是随着函数执行而产生，并随着函数结束而消失的。为什么这里 p 引用的函数 printer()能够访问函数 print_msg()的局部变量呢? 这是因为，当函数 print_msg()返回其嵌套函数对象时，会一起将其局部变量 msg 和函数 printer()打包在一起，因此返回的不仅是函数对象 printer()本身，而且实际是一个包裹了函数 printer()和其包围环境的变量(如 msg)的包裹对象，即返回的是一个闭包对象。

所谓**闭包**，就是一个函数对象记住了其包围环境(enclosing scope)的一些信息。

一个闭包必须满足如下条件。

- 必须有一个嵌套函数(定义在其他函数内部的函数)。
- 嵌套函数必须引用其包围环境中的一个值。
- 包围函数必须返回嵌套函数。

8.2.4 用闭包代替类

嵌套函数能够使用其包围函数中的值，也就是说，将嵌套函数和其包围环境中的数据打包在一

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

起，这类似于面向对象中用类将数据和方法打包在一起。因此，对于只包含一个函数的类，完全可以用闭包来代替类。例如：

```
def Exp(a):
    def Power(b):
        return a**b
    return Power      #返回的是嵌套函数 Power()的闭包

exp_10 = Exp(10)
```

函数 Exp()返回的是嵌套函数 Power()的闭包，即返回的函数对象 Power()记住了其包围函数 Exp()中的局部变量(如 a)。“exp_10 = Exp(10)”返回的函数对象 Power()记住了其包围函数 Exp()中的变量 a 的值是 10。例如：

```
print(exp_10(3.0))
```

输出：

```
1000.0
```

调用函数 exp_10()引用的函数对象 Power()并传递参数值 3.0，该函数用记住的 a 值 10 作为底数，而实参 3.0 作为形参 b 的值，计算指数，返回指数运算的结果。再看下面代码：

```
import math
exp_e = Exp(math.e)

print(exp_e(3.0))
print(exp_e(5.0))
```

输出：

```
20.085536923187664
148.41315910257657
```

函数 exp_e()引用的函数对象 Power()记住了包围函数中的 a 值是 math.e，然后用它引用的这个函数计算底是 e 而指数分别是 3.0 和 5.0 的指数函数值。

因此，对于只有一个函数的类或一个函数需要使用其包围环境中的数据时，可考虑使用闭包。

8.2.5 函数的闭包属性__closure__

实际上，所有函数都有一个闭包属性__closure__，如果这个函数是一个闭包，那么它返回的是一个由 cell 对象组成的元组对象。cell 对象的 cell_contents 就是闭包中的数据。例如：

```
Exp.__closure__
```

Exp 的属性 __closure__ 是空值，而 exp_e 的属性__closure__则不是空值：

```
exp_e.__closure__
```

输出：

```
(<cell at 0x000001F1EB64A2E8: float object at 0x000001F1E71229D8>,)
```

输出其下标为 0 的第一个 cell 中的 cell_contents，就是自然底数 e 的值：

```
exp_e.__closure__[0].cell_contents
```

输出：

```
2.718281828459045
```

总结

- 介绍作用域、嵌套函数和闭包的概念。
- 闭包就是返回的一个嵌套函数对象，而这个嵌套函数对象使用了其包围环境中的数据，这

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

些数据被打包在这个返回的嵌套函数对象里。

- 每个函数都有一个闭包属性__closure__，嵌套函数的闭包属性__closure__保存的是其包围函数中的局部数据。

8.3 装饰器

前面讲过，Python 中一切都是对象，函数也是对象，函数名可以看作引用函数对象的一个变量名，可以将函数复制给一个变量，一个函数可作为其他函数的参数或返回值（函数是第一类对象（First Class Objects）），可以在函数里定义嵌套函数。

作为 Python 中难懂但也是很重要的一个概念——装饰器（decorators）也是一个函数或类，它用于给一个已有的函数添加一些额外的辅助功能，如添加日志功能、性能测试、事务处理、缓存、权限校验等。

1. 什么是装饰器

假如有如下程序：

```
def say():
    print("这里是教小白精通编程!")
def hello():
    print("hello!")
```


如果要为程序的多个函数（如以上代码中的 say()和 hello()函数）添加日志功能，那么最容易想到的是修改这些函数添加记录日志的功能。因为日志是记录到文件中的，所以这里用 print()模拟记录日志：

```
import time

def say():
    print('say() start at %s' % time.time())
    print("这里是教小白精通编程!")
    print('say() stopped at %s' % time.time())

def hello():
    print('say() start at %s' % time.time())
    print("hello!")
    print('say() stopped at %s' % time.time())

say()
hello()
```

以上为每个函数增加了开始和结束时间的日志功能，程序输出如下：

```
say() start at 1532834458.0996063
这里是教小白精通编程!
say() stopped at 1532834458.0996063
say() start at 1532834458.0996063
hello!
say() stopped at 1532834458.0996063
```

这种方式需要修改每个需要添加日志的函数，如果以后又要修改这段日志代码，则这些函数里的代码就都需要逐个修改，这样既破坏了原有的函数，又不能复用（重复使用）记录日志的代码，很不方便。

为了克服这些缺点，可以另外定义一个接收函数作为参数的包裹函数，然后将日志记录的代码放在这个包裹函数里，而且需要记录日志的函数可以作为参数传给这个函数。例如：

```
def wrapper(func):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    print('say() start at %s' % time.time())
    func()
    print('say() stopped at %s' % time.time())
```

函数 wrapper()接收另一个函数作为其参数，函数 wrapper()的开始和结束语句增加了记录开始和结束时间的日志功能。现在可以用这个函数 wrapper()包裹函数 say()和函数 hello()。例如：

```
def say():
    print("这里是教小白精通编程!")

def hello():
    print("hello!")

wrapper(say)                #say()函数传递给 wrapper()的形参 func
wrapper(hello)              #hello()函数传递给 wrapper()的形参 func
```

输出：

```
say() start at 1532834462.6293955
这里是教小白精通编程!
say() stopped at 1532834462.6293955
say() start at 1532834462.6293955
hello!
say() stopped at 1532834462.6293955
```


日志代码放在函数 wrapper()里，并将函数 say()或函数 hello()作为参数传给这个函数，从而给函数 say()和函数 hello()添加日志记录的功能。编写一次日志就可以多次重复使用(给任意多的函数添加同样的日志功能)，也更容易维护，因为只要保证具有这个日志功能的包裹函数正确就可以了。

但是，程序中原来调用函数 say()或函数 hello()的地方都需要修改为调用函数 wrapper(say)、函数 wrapper(hello)，所以破坏了原有代码。

有没有更好的方法呢？答案就是：“装饰器”。将上面的函数 wrapper()作为一个外部函数 decorator()的返回值，这个外部函数 decorator()称为**装饰器函数**。例如：

```
def decorator(func):
    def wrapper():
        print('say() start at %s' % time.time())
        func()
        print('say() stopped at %s' % time.time())

    return wrapper


def say():
    print("这里是教小白精通编程!")

def hello():
    print("hello!")

say = decorator(say)            #返回包裹了函数 say()的函数 wrapper()，但并没有执行它
hello = decorator(hello)        #返回包裹了函数 hello()的函数 wrapper()，但并没有执行它
say()                           #执行包裹了函数 say()的函数 wrapper()
hello()                         #执行包裹了函数 hello()的函数 wrapper()
```

输出：

```
say() start at 1532834638.1917906
这里是教小白精通编程!
say() stopped at 1532834638.1927903
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
say() start at 1532834638.1927903
hello!
say() stopped at 1532834638.1927903
```

将函数 say()作为装饰器函数的参数，而装饰器函数返回了包裹函数 wrapper(),并重新命名为 say，程序中调用函数 say()的代码不需要做任何修改，即自动添加了额外的功能。

每次采用“say = decorator(say)”形式，利用装饰器函数装饰被装饰的函数，显得有些麻烦，Python 用一个特殊的符号@来达到同样的效果，即@符号后面加上装饰器函数的名字(如这里的 decorator)：

```
@decorator
def say():
    print("这里是教小白精通编程!")

@decorator
def hello():
    print("hello!")
```

然后就可以使用被装饰的函数 say()或函数 hello()了：

```
say()
hello()
```


输出：

```
say() start at 1529572974.2849686
这里是教小白精通编程!
say() stopped at 1529572974.2859797
say() start at 1529572974.2859797
hello!
say() stopped at 1529572974.2859797
```

2. 向被装饰的函数传递参数

首先看下面的代码：

```
@decorator
def say(name):
    print(name,"这里是教小白精通编程!")

@decorator
def hello(name):
    print(name,"hello!")

say('wang')
print()
hello('张')
```

执行抛出 TypeError 异常“wrapper() takes 0 positional arguments but 1 was given”：

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-16-36e61000e022> in <module>()
      7     print(name,"hello!")
      8
----> 9 say('wang')
     10 print()
     11 hello('张')
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
TypeError: wrapper() takes 0 positional arguments but 1 was given
```

这是因为，decorator 里的 wrapper()不带参数，却给被装饰的函数 say()传递了一个参数“say('wang')”。解决这个问题的方法是给包裹器函数 wrapper() 添加参数，由包裹器函数 wrapper()将这个参数传递给被包裹的函数 func()：

```
def decorator(func):
    def wrapper(Name):
        print('say() start at %s' % time.time())
        func(Name)
        print('say() stopped at %s' % time.time())

    return wrapper

@decorator
def say(name):
    print(name,"这里是教小白精通编程!")

@decorator
def hello(name):
    print(name,"hello!")

say('wang')
print()
hello('张')
```


输出：

```
say() start at 1529573070.6684735
wang 这里是教小白精通编程!
say() stopped at 1529573070.6684735

say() start at 1529573070.6684735
张 hello!
say() stopped at 1529573070.6694772
```

上面的装饰器给不同的函数 say()和 hello()添加的都是完全一样的功能。如何使添加的功能能够根据情况做一些变化呢？答案就是：对装饰器函数再做一次闭包的封装。首先将装饰器函数作为另外一个函数的嵌套函数，并在该函数里返回这个装饰器函数。这个返回的装饰器函数是一个闭包，可以访问其外层函数的参数。然后用这个外层函数作为@符号后的装饰器名字。例如：

```
def decorator_wrapper(arg,arg2):
    def decorator(func):
        def wrapper(Name):
            print(arg)
            func(Name)
            print(arg2)

        return wrapper
    return decorator

@decorator_wrapper(arg="welcome",arg2="have a good time")
def say(name):
    print(name,"这里是教小白精通编程!")

@decorator_wrapper(arg="你好.",arg2="希望你学习愉快")
def hello(name):
    print(name,"hello!")
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
say('wang')
print()
hello('张')
```

输出：

```
welcome
wang 这里是教小白精通编程!
have a good time

你好.
张 hello!
希望你学习愉快
```

通过给装饰器的包裹函数传递不同参数，就可以使装饰器具有不同的装饰功能。当然，和普通函数一样，装饰器里的包裹函数和被包裹函数也可以接收可变参数和字典参数。例如：

```
def decorator_wrapper(arg,arg2):
    def decorator(func):
        def wrapper(*args,**kwargs):          #包裹函数可以接收可变参数和字典参数
            print(arg)
            func(*args, **kwargs)             #被包裹函数也接收可变参数和字典参数
            print(arg2)

        return wrapper
    return decorator

@decorator_wrapper(arg="welcome",arg2="have a good time")
def say(name,*args, **kwargs):
    '''say 函数 '''
    print(name)
    print('可变参数: ',end = ' ')
    for e in args:
        print(e ,end = ',')
    print('\n 字典参数: ',end = ' ')
    for key in kwargs:
        print(key,kwargs[key],end = ',')
    print()

say('wang',23,'shanghai',python = 67,ds=80)
```

输出：

```
welcome
wang
可变参数: 23,shanghai,
字典参数: python 67,ds 80,
have a good time
```

3. 类装饰器

装饰器不仅可以是一个函数，也可以是一个类，也就是用一个装饰器类去包裹一个函数，从而给函数添加功能。类装饰器主要依靠类的__call__()方法，当使用@形式将装饰器附加到函数上时，就会调用此方法。例如：

```
class decorator(object):
    def __init__(self, func):
        self._func = func
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    def __call__(self,name,*args,**kwargs):
        print ('class decorator start')
        return self._func(name,*args,**kwargs)


@decorator
def say2(name,*args, **kwargs):
    print(name)
    print('可变参数: ',end = ' ')
    for e in args:
        print(e ,end = ',')
    print('\n 字典参数: ',end = ' ')
    for key in kwargs:
        print(key,kwargs[key],end = ',')
    print()

say2('wang',23,'shanghai',python = 67,ds=80)
```

输出：

```
class decorator start
wang
可变参数: 23,shanghai,
字典参数: python 67,ds 80,
```


4．functools.wraps

上面的装饰器还有一些缺点，即会导致原函数的一些属性(元信息)不正常，如函数的__doc__属性、__name__属性。例如：

```
print(say.__name__)
```

输出：

```
wrapper
```

此时，say 的 name 变为函数 wrapper()的函数名，解决方法是使用 functools.wraps。wraps 本身也是一个装饰器，它能够把原函数的元信息复制到装饰器里面的函数 func()中，这使得装饰器里面的函数 func()也有和原函数 say()一样的元信息了。例如：

```
from functools import wraps       #导入 functools.wraps

def decorator_wrapper(arg,arg2):
    def decorator(func):
        @wraps(func)                #原函数的元信息复制到装饰器里的函数 func()中
        def wrapper(*args,**kwargs):
            print(arg)
            func(*args, **kwargs)
            print(arg2)

        return wrapper
    return decorator

@decorator_wrapper(arg="welcome",arg2="have a good time")
def say(name,*args, **kwargs):
    '''say 函数 '''
    print(name)
    print('可变参数: ',end = ' ')
    for e in args:
        print(e ,end = ',')
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        print('\n字典参数：',end = ' ')
        for key in kwargs:
            print(key,kwargs[key],end = ',')
        print()

    say('wang',23,'shanghai',python = 67,ds=80)
```

输出：

```
welcome
wang
可变参数：23,shanghai,
字典参数：python 67,ds 80,
have a good time
```

被包裹后的函数 say()的属性__name__就正确了：

```
print(say.__name__)
```

输出：

```
say
```

总结

- 装饰器就是一个返回对传入的函数进行包裹的包裹函数或类，可以用于对输入的函数添加一些额外的功能，如日志记录、性能测试等。
- 装饰器装饰的函数就是一个普通函数，可以带有可变参数和字典参数，而装饰器本身也可以带有参数。
- 类装饰器是通过__call__属性来执行包裹功能的。
- functools.wraps 也是一个装饰器，用于将被包裹函数的元信息(如__doc__属性、__name__属性等)传递给包裹函数。

8.4 @property

在面向对象的编程过程中，为了保证数据封装效果，一个类的数据通常是私有的，即外界不能随便访问一个类对象的数据，如果外界可以任意访问或修改数据，则会造成数据被无意或有意地破坏。例如：

```
class Employee:
    def __init__(self,name,salary):
        self.name = name
        self.salary = salary
    def printInfo(self):
        print(self.name,",",self.salary)
```

可以创建 Employee 对象，并访问它的数据。例如：

```
e  = Employee('Wang',1000)
e.salary= 8000
e.printInfo()
```

输出：

```
Wang , 8000
```

假如一个 Employee 对象被很多客户程序或函数访问，则有可能会无意或有意地使对象数据处于异常状态。例如，salary 是一个负数，而实际中 salary 应该是大于等于“0”的。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

为了保证数据的安全性，一个类的数据经常是私有的，即外界不能随意修改它，只能通过该类自身提供的公开接口，即公开方法去访问或修改类对象的数据。为了提供外界查询或修改类对象的数据，需要在类定义中给这些数据增加相应的 getters 和 setters 方法。例如：

```
class Employee:
    def __init__(self,name,salary=0):
        self.set_name(name)
        self.set_salary(salary)
    def printInfo(self):
        print(self.__name,",",self.__salary)
    def get_name(self):
        return self.__name
    def set_name(self,name):
        self.__name = name

    def get_salary(self):
        return self.__salary
    def set_salary(self,salary):
        if(salary<0):
            self.__salary = 0
        else:
            self.__salary = salary
```


上述的函数 set_salary() 可以检查输入参数 salary 是否合法，如果小于“0”，则将对象的__salary 属性设置为“0”，从而保证数据的安全合法性。任何试图直接访问类对象的属性的企图都是无效的。例如：

```
e  = Employee('Wang',1000)
e.__salary= 8000
e.printInfo()
```

输出：

```
Wang , 1000
```

只能通过类提供的接口方法去访问它。例如：

```
e.set_salary(8000)
e.printInfo()
e.set_salary(-200)
e.printInfo()
```

输出：

```
Wang , 8000
Wang , 0
```

用这种 getters 和 setters 方法当然可以保证对象数据的有效合法性，但需要将原先直接访问对象的属性的代码都换为函数调用代码，但这样做不如直接访问属性方便。

解决这一问题的技术是使用 Python 的函数 property()。

1．函数 property()

为了能够直接通过**成员访问运算符.**访问类对象的私有数据，则函数可以借助 property()，该函数将直接通过成员访问运算符.访问类对象私有数据的操作转为对应的公开的 getter 方法或 setter 方法。例如：

```
class Employee:
    def __init__(self,name,salary=0):
        self.set_name(name)
        self.salary = salary
    def printInfo(self):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            print(self.__name,",",self.__salary)
        def get_name(self):
            return self.__name
        def set_name(self,name):
            self.__name = name

        def get_salary(self):
            return self.__salary
        def set_salary(self,salary):
            if(salary<0):
                self.__salary = 0
            else:
                self.__salary = salary

        salary = property(get_salary,set_salary)
```

由上段程序可见，在类的最后添加了一行代码“salary = property(get_salary, set_salary)”，并在构造函数 init 里设置“ self.salary = salary”。

现在就可以采用如下方法：

```
e  = Employee('Wang',1000)
e.salary = 8000
e.printInfo()
e.salary = -200
e.printInfo()
```


输出：

```
Wang , 8000
Wang , 0
```

注意

构造函数__init__()里的“self.salary = salary”实际上触发了函数调用 set_salary()。当然，对于 name 数据属性也可以这样做(作为练习)。

函数 property()是 Python 的一个内置函数，它创建并返回了一个 property 对象，该函数的格式是：

```
property(fget=None, fset=None, fdel=None, doc=None)
```

其中，fget、fset、fdel 分别用来获得、修改、删除一个数据属性，doc 是一个文档串。这些函数参数都是可选的，可以不带任何参数调用这个函数。例如：

```
property()
```

返回的是一个 property 对象，这个对象有三个方法，即 getter()、setter()和 delete()，用于指定传入的 fget、fset、fdel 函数。

因此，上述代码中的语句：

```
salary = property(get_salary,set_salary)
```

实际相当于：

```
salary = property()
salary = salary.getter(get_salary)          #赋值 fget
salary = salary.setter(set_salary)          #赋值 fset
```

2．@property

可以看出，函数 property()就是装饰器。

既然如此，就完全可以用@property 来包裹需要包裹的方法了：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
class Employee:
    def __init__(self,name,salary=0):
        self.__name = name
        self.__salary = salary

    def printInfo(self):
        print(self.__name,",",self.__salary)

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self,name):
        self.__name = name

    @property
    def salary(self):
        return self.__salary

    @salary.setter
    def salary(self,salary):
        if(salary<0):
            self.__salary = 0
        else:
            self.__salary = salary
```

执行下列代码：

```
e  = Employee('Wang',1000)
e.salary = 8000
e.printInfo()
e.salary = -200
e.printInfo()

e.name = 'Li Ping'
e.printInfo()
```

输出：

```
Wang , 8000
Wang , 0
Li Ping , 0
```

注意

get_name()和 set_name()等方法名都改为了 name()，并分别在函数名前面添加了@property 和@name.setter。对于 salary 也可采用类似方法处理。

总结

- getters()和 setters()是面向对象为了达到数据封装效果的常用方法，可以用这些方法作为接口去访问私有数据属性。
- 函数 property()是一个装饰器函数，通过装饰器函数 property()包裹了相应的数据属性 getters()和 setters()方法，从而使其可以直接通过成员访问运算符.去访问这些属性。
- @property 比直接调用函数 property()更加方便。对于某个数据属性的 getter()方法，在前面添加@property，就可以直接查询类对象的这个方法对应的数据属性，而对于某个数据

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

属性的 setter()方法，在前面添加@xxx.setter，就可以直接修改类对象的这个方法对应的数据属性。

8.5 类的静态方法和类方法

前面介绍的类的方法都是实例方法，即必须通过一个类的具体实例(对象)才能调用类的实例方法。实例方法必须通过第一个 self 形参绑定调用它的具体实例。因此，实例方法不能通过“**类名.实例方法**(...)”这种形式调用。

类中还可以定义和类相关的**类方法**和**静态方法**，这两种方法和类的具体实例无关，因此既可以通过类名也可以通过具体实例名调用这两种方法。这两种方法分别用装饰器@staticmethod 和@classmethod 开头。

8.5.1 静态方法

类的静态方法用装饰器@staticmethod 开头，没有任何和类或类的实例有关的隐式参数(如 self)。定义格式如下：

```
class C:
    @staticmethod
    def static_f(arg1, arg2, ...):
      ...
```


类的静态方法和普通的全局函数，或者说，模块的函数没有任何区别，完全可以用普通的模块中的函数代替类的静态方法，即将上述的静态方法写为一个普通的外部函数：

```
def static_f(arg1, arg2, ...):
...
```

那么，为什么要写为类的静态方法呢？这主要是因为这个方法所做的事情和这个类有关系，如初始化与某个类相关的数据。将这种和某个类有关的函数放在类的内部成为类的静态方法有如下三个优点。

- 告诉其他人这个方法是和这个类有些关系的。
- 不需要通过“模**块名.函数名**”或“**单独导入函数名**”的方式来使用这个函数。
- 使代码组织清晰，便于维护、管理和跟踪。

例如，str 类有一个静态方法 maketrans()可以生成一个字符的翻译表，而 str 类的其他方法如 translate()方法就需要这种翻译表来对字符串进行翻译转换。例如：

```
trantab = str.maketrans("aeiou", "12345") #构造元音字母到数字的翻译表
s = "this is string example ... wow!!!"
print(s.translate(trantab))    #将字符串 s 中的元音字母"aeiou"转换为数字"12345"
```

输出：

```
th3s 3s str3ng 2x1mpl2 ... w4w!!!
```

str 类的 maketrans()方法是一个只和 str 类有关的方法，将它定义为 str 类的静态方法是很自然的。

类的静态方法既可以通过类名访问，也可以通过类的实例去访问。例如：

```
trantab = '.'.maketrans("aeiou", "12345") #构造元音字母到数字的翻译表
```

8.5.2 类方法

类的类方法用装饰器@classmethod 开头，其第一个参数是表示类本身的隐式参数 cls。定义格式如下：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
class C:
    @classmethod
    def class_f(cls,arg1, arg2, ...):
        ...
```

类的静态方法处理和类相关的数据，但不能访问或修改类本身的状态(数据)。而类的类方法通常用于访问或修改类本身的状态(数据)，如处理类的类属性或用于创建一个类的实例。例如，前面讲解的表示日期的 Date 类，其构造函数接收三个 int 型的表示年、月、日的参数：

```
class Date:
    default_date = [2018,1,1]
    def __init__(self, year=default_date[0],month=default_date[1],day=default_date[2]):
        self.year = year
        self.month = month
        self.day = day
    def Print(self):
        print(self.year,'-',self.month,"-",self.day)
```

假如现在传递的是一个字符串格式的日期参数“2018-8-18”，则需要首先字符串从中分离出表示单独的年、月、日的数值，再用构造函数创建一个 Date 实例：

```
s_date = '2018-8-18'
year,month,day = map(int,s_date.split('-'))
d = Date(year,month,day)
d.Print()
```

输出：

```
2018 - 8 - 18
```

但是，每次进行这样的处理有点烦琐，因此可以将这个处理功能定义为 Date 的类方法 create_date()：

```
class Date:
    default_date = [2018,1,1]
    def __init__(self, year=default_date[0],month=default_date[1],day=default_date[2]):
        self.year = year
        self.month = month
        self.day = day
    def Print(self):
        print(self.year,'-',self.month,"-",self.day)

    @classmethod
    def create_date(cls,string):
        #第一个参数是 cls，表示这个类
        year,month,day = map(int,string.split('-'))
        return cls(year,month,day )
```

实例方法的第一个参数必须是 self，用于指向调用这个方法的实例的引用，类的类方法第一个参数必须是 cls，用于指向调用这个类方法的类。可以通过类名或类的实例调用这个类方法，以创建一个 Date 类的对象(实例)：

```
s = '2018-8-18'
d = Date.create_date(s)
d.Print()
s2 = '2016-6-6'
d2 = d.create_date(s2)
d2.Print()
```

输出：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
2018 - 8 - 18
2016 - 6 - 6
```

总结

- 类的方法有实例方法、静态方法和类方法。实例方法的第一个参数必须是 self，用于指向调用这个方法的类的实例。而类方法用@classmethod 装饰器来创建，其第一个参数必须是 cls，用于指向调用这个方法的类本身。静态方法用@staticmethod 装饰器来创建，和普通的外部函数一样，没有任何隐式参数。
- 类的静态方法通常处理和类相关的数据，而类的类方法通常处理类本身的数据。
- 实例方法只能通过类的实例来调用，而类的静态方法和类方法既可以通过类名也可以通过类的实例来调用。

8.6 浅拷贝、深拷贝

首先回顾几个概念。

- 变量仅是对象的名字(引用)，对变量赋值就是给对象起名字(用变量名引用实际对象)。
- 用一个变量 a 给另一个变量 b 赋值，实际就是让 b 和 a 都引用同一个对象。
- 用一个对象对另一个变量赋值，实际就是让这个变量引用这个新的对象。

例如：

```
#用一个变量 a 给另一个变量 b 赋值，实际就是让 b 和 a 都引用同一个对象
a = 34                  #a 引用了对象 34
b = a                   #b 引用了 a 引用的对象，即 34
print(b is a)

print(id(a))
print(id(b))
```

输出：

```
True
1549955168
1549955168
```

因此，对一个变量赋值就是让该变量引用一个对象。“b=a”就是让 b 引用 a 引用的那个对象，即 a 和 b 引用的是同一个对象。

可以随时让变量指向(引用)另外的对象。例如：

```
#用一个对象对一个变量赋值，实际是让该变量引用这个新的对象
a = [34]                #a 引用一个 list 对象[34]
b = a                   #b 和 a 一样引用同一个对象
print(id(a))
a = 'hello'             #a 引用了新的 str 对象'hello'
print(id(a))            #新对象的地址
print(id(b))            #旧对象的地址
```

输出：

```
1930580367816
1930581599096
1930580367816
```

如果两个变量引用的是同一个可变的对象，则通过任意一个变量修改引用的那个对象其效果都是一样的。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
a= ['hello',[3],[5,6,8]]
b = a
b[0] = 29
print(a)
print(b)
```

输出：

```
[29, [3], [5, 6, 8]]
[29, [3], [5, 6, 8]]
```

因此，赋值运算符仅是让变量引用新的对象，而不是通常意义上的**复制**(也称**拷贝**)操作。但是，如果希望一个新对象复制(copy)原来的对象，则新对象和旧对象是相互独立的，对其中任意一个对象的修改都不会影响另外的对象。显然，采用简单的赋值语句是不可行的。

Python 提供了以下两种复制(拷贝)对象的方式。

- 浅拷贝(Shallow copy)：创建一个新的对象，则新的对象包含的元素和原有的对象的对应元素是同一个嵌套对象的引用。尽管新的对象和原有的对象不一样，但是它没有创建嵌套的子对象的复制(拷贝)。
- 深拷贝(Deep copy)：创建一个新的对象，原有的对象包含的元素引用的子对象如果是可变的，则新的对象的对应元素引用的是这个子对象的深拷贝。这个深拷贝过程是一直递归的，直到嵌套子对象是不可变的为止。

假如有一份学生名单，然后复制了一份学生名单，这种复制的学生名单就是“浅拷贝”，因为这是两份独立的学生名单，但两份名单上的同一个名字指向的学生是同一人，并没有“克隆”出一模一样的另外一个人。

再如，复制一个网页文件，产生的新的网页文件和原来的网页文件是两个不同的对象，但它们的内容是完全一样的，两个网页的对应的超链接的值都是一样的，即都是指向同一个资源。也就是说，两个内容完全相同的网页虽然是两个不同的文件，但它们的对应超链接指向的每个资源却只有一个。

8.6.1 浅拷贝

浅拷贝类似复制一个网页文件，产生的新的网页文件和原来的网页文件是两个不同的对象，但它们的内容是完全一样的，两个网页的对应的超链接的值都是一样的，即都是指向同一个资源。也就是说，两个内容完全相同的网页虽然是两个不同的文件，但是它们的对应的超链接指向的每个资源却是同一个。

Python 的 copy 模块的函数 copy()提供了对象的浅拷贝操作：

```
copy.copy(x)
```

例如，下面的变量 b 就是 a 的浅拷贝，二者是完全不同的对象：

```
import copy
a= ['hello',3,[3,5,6]]
b = copy.copy(a)
print(b is a )
print(a)
print(b)
print(id(a[1]), '\t',id(a[2]))
print(id(b[1]), '\t',id(b[2]))
```

输出：

```
False
['hello', 3, [3, 5, 6]]
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
['hello', 3, [3, 5, 6]]
140735891739520   2203097966216
140735891739520   2203097966216
```

如图 8-3 所示，尽管 a、b 是不同的对象，但是两个对象对应元素的值是一样的，引用的是同一个对象：

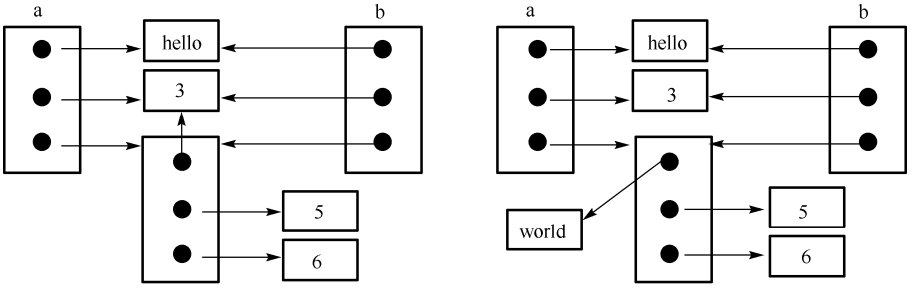
```
print(b[0] is a[0] )   #b[0]和a[0]的值是一样的，都是同一个对象的引用
print(id(a[0]))
print(id(b[0]))
```

输出：

```
True
1391992521984
1391992521984
```

可以看到，a 和 b 引用的是不同的对象，但两个 list 对象的对应元素引用的仍然是同一个对象。既然引用的是同一个对象，那么，可以通过一个元素去修改这个引用的嵌套子对象(假设这个子对象是可以修改的)，也就是修改了对应元素引用的同一个嵌套子对象。

```
b[2][0]='world'       #修改b[2]，也就是修改了a[2]，因为它们是同一个对象的引用
print(a)
print(b)
print(id(b[2]))
print(id(a[2]))
```

(a) 浅拷贝　　(b) b[2][0] = “world”

图 8-3　浅拷贝

输出：

```
['hello', 3, ['world', 5, 6]]
['hello', 3, ['world', 5, 6]]
2203097966216
2203097966216
```

但是，如果 list 的元素引用的是不可修改的嵌套对象，则无法修改这个对象：

```
b[0][0]='W'
```

抛出 TypeError 异常：

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-b963f1a155ca> in <module>()
----> 1 b[0][0]='W'

TypeError: 'str' object does not support item assignment
```

如果直接给其中一个 list 对象的元素用一个新的对象赋值，则这个 list 的这个元素就引用这个新

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

的对象，而原来的 list 的对应元素仍然是引用原来的嵌套对象。因此，这个时候，两个 list 对象的元素的值就不一样了(因为其中的对应元素的引用不一样了)：

```
b[0] = 29
print(a)
print(b)
```

输出：

```
['hello', 3, ['world', 5, 6]]
[29, 3, ['world', 5, 6]]
```

同样，可以修改 b[2]，让它引用新对象，而 a[2]仍然引用的是原来的对象：

```
b[2] = [17]
print(a)
print(b)
```

输出：

```
['hello', 3, ['world', 5, 6]]
[29, 3, [17]]
```

因此，假如一个对象是嵌套的，即对象内部包含其他对象的引用，那么浅拷贝只对最外层的对象包含的元素值进行复制(拷贝)，而没有对这些元素值引用的嵌套对象进行复制(拷贝)：

```
import copy
a= ['hello',3,[3,5,6]]
b = copy.copy(a)

for e in a:
   print(id(e))
print()
for e in b:
   print(id(e))
```

输出：

```
1391992521984
1596353664
1391992466248

1391992521984
1596353664
1391992466248
```

可以看出，两个 list 对象的对应元素的 id 值都是一样的(引用的是同一个对象)。

8.6.2 深拷贝

和浅拷贝不同，对一个对象进行深拷贝会对这个对象的嵌套对象进行复制(拷贝)，并且对嵌套对象的嵌套对象进行复制(拷贝)，这个过程会一直深入下去，直到所有深度的嵌套对象都得到了复制(拷贝)。

Python 的 copy 模块的函数 deepcopy()提供了对象的深拷贝操作：

```
copy.deepcopy(x, memo=None, _nil=[])
```

例如，下面用深拷贝，对对象 a 引用的 list 对象进行复制，并且对嵌套子对象也重复这个复制(拷贝)过程。因此，新旧对象的对应元素引用的就不再是同一个嵌套子对象了。可以通过检查对应元素的 id 来验证这一点：

```
import copy
a= ['hello',3,[3,5,6]]
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
b = copy.deepcopy(a)

for e in a:
    print(id(e))
print()
for e in b:
    print(id(e))
```

输出：

```
1391992521984
1596353664
1391992465224

1391992521984
1596353664
1391992592072
```

而深拷贝会一直对可变的嵌套子对象进行递归的复制(拷贝)，不可变的嵌套子对象如“hello”和“3”仍然是共享的，如图 8-4 所示。

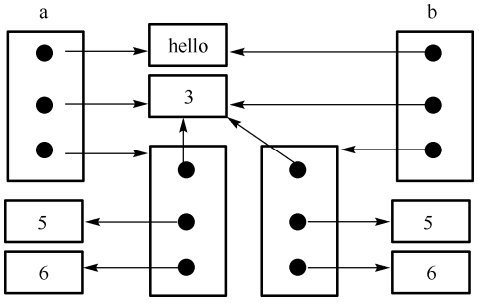


图 8-4　深拷贝

对任意一个对象的可变嵌套对象进行修改不会影响其深拷贝对象的对应的嵌套子对象，因为它们是独立的不相干的对象。例如：

```
b[2][0]='world'  #修改b[2]不会影响a[2]
print(a)
print(b)
```

输出：

```
['hello', 3, [3, 5, 6]]
['hello', 3, ['world', 5, 6]]
```

总结

- 浅拷贝(copy.copy())返回的新的对象和原有的对象的对应元素的值是完全相同的，是同一个嵌套对象的引用，即浅拷贝不复制(拷贝)嵌套的子对象。
- 深拷贝(copy.deepcopy())返回的新的对象是对原有的对象及其可变的嵌套子对象的递归复制(拷贝)，但不可修改的子对象仍然是共享的，即不会复制(拷贝)。

8.7　习题

1．如何从一个 list 对象创建一个迭代器对象？(　　)

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

A．将 list 对象传递给函数 iter()　　B．用一个 for 循环
C．用一个 while 循环　　D．无法从 list 对象创建一个可迭代对象

2．包含了 yield 语句的函数是(　　)。

A．可迭代的　　B．生成器函数
B．匿名函数　　D．上述说法都不对

3．在以下程序段中的问号?处补充代码，以完善生成器函数，使其可以生成一对整数(low 和 high)之间的所有整数。

```
def counter(low, high):
    current = low
    while current <= high:
        ?
for c in counter(3, 8):
    print c
```

4．下列程序的输出是(　　)。

```
my_list = [1, 3, 6, 10]
a = (x**2 for x in my_list)
print(next(a), next(a))
```

A．1 3　　B．1 9　　C．1 9 36 100　　D．1

5．编写一个程序，接收一个或多个文件名作为参数，输出文件中字符个数少于 60 个的那些行。

6．编写一个函数，输入一个整数“n”和一个文件名，将这个文件分割成每个文件都是“n”行的多个文件(除最后一个文件可以少于“n”行)。

7．用如下的代码来装饰 8.3 节中的函数 say()和函数 hello()，从而给函数 say()和函数 hello()添加功能。

```
wrapper(say)
wrapper(hello)
```

但是调用函数 say()和函数 hello()的代码都要替换为上面的调用形式，为此，在函数 wrapper()外层又定义了 decorator()函数来返回装饰函数 say()和函数 hello()，使得调用函数 say()和函数 hello()的代码不会受到任何影响。

```
say = decorator(say)
hello = decorator(hello)
```

请解释为什么不可以直接用如下形式包裹函数 say()和函数 hello()。

```
say = wrapper(say)
hello = wrapper(hello)
```

8．以下哪个函数是装饰器函数？(　　)

```
def mk(x):
    def mk1():
        print("Decorated")
        x()
    return mk1
def mk2():
    print("Ordinary")
p = mk(mk2)
p()
```

A．p()　　B．mk()　　C．mk1()　　D．mk2()

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

9．假如 f 是一个装饰器函数，那么下面两个代码片段是否是等价的？（　　）

片段 1：

```
@f
def f1():
    print("Hello")
```

片段 2：

```
def f1():
    print("Hello")
f1 = f(f1)
```

A．是　　B．否

10．下列代码的输出是（　）。

```
def d(f):
    def n(*args):
        return '$' + str(f(*args))
    return n
@d
def p(a, t):
    return a + a*t

print(p(100,0))
```

A．100　　B．$100　　C．$0　　D．0

11．运行下列代码，并给函数 student()添加位置参数(name、age)、可变参数(address、class、mobile)和字典参数(每门课对应的成绩：{‘C’:80,’Python’:90,’DS’:70.8})，并在函数 student()中输出这些参数。

```
def make_bold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped

def make_italic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped

def make_underline(fn):
    def wrapped():
        return "<u>" + fn() + "</u>"
    return wrapped

@make_bold
@make_italic
@make_underline
def student():
    return "a student"
print(student())    #输出："<b><i><u>a student</u></i></b>"
```

12．定义表示日期的 date 类，对其中的属性的读/写都采用@property。

13．请为上述的 date 类编写一个类方法，用于修改 date 类的类属性 default_date。

14．结合实际生活中的例子说明“深拷贝”的含义。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Python 标准库

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 9 章　标准库的常用模块

Python 受广泛欢迎的一个重要原因是它拥有大量的库(模块/包)，它既有 Python 自带的标准库，也有第三方开发的各种库，正是由于这些大量的库，使得程序员可以避免重复造轮子，因此极大地提高了程序的开发效率。在 Python 中，无论是标准库还是第三方库数量都非常多，这些库不可能也不需要都学习和了解，本书只选取少量常用的库来说明这些库是如何提高程序开发效率的。相信具有 Python 基本编程知识的读者都可以通过上网搜索学会并使用更多的 Python 库。

前面已经讲解过 Python 标准库的一些模块，如用于数学计算的 math，与解释器交互的 sys 模块，产生随机数的 random 模块，I/O(输入/输出)模块。

本章和下一章将对标准库的一些常用模块进行简要介绍。

9.1　操作系统接口模块

9.1.1　os 模块

1．os 模块中常用函数和常量

os 模块是提供操作系统相关功能的模块，如 os 和 os.path 中包含了操作目录、文件等相关的函数。若要使用 os 模块，则需要首先导入它：

```
import os
```

以下介绍 os 模块(包括 os.path 模块)中的一些常用函数或常量。

os.**name**：指示用户正在使用的工作平台。例如，对于 Windows 用户，它是'nt'，而对于 Linux/Unix 用户，它是'posix'。例如：

```
>>> import os
>>> os.name
'nt'
```

os.**getenv**(name, default=None)：读取环境变量 name 的值，如果不存在则返回 None，该函数可以传递 default 参数作为不存在的返回值。例如：

```
>>>os.getenv("TMP")
'C:\\Users\\s\\AppData\\Local\\Temp'
```

os.**putenv**(name, value)：改变或添加环境变量 name 的值为 value。

os.**getcwd**()：返回表示当前工作目录的 unicode 字符串。例如：

```
>>>os.getcwd()
'D:\\Python\\Python37'
```

os.**chdir**(path)：改变工作目录到指定的路径 path，path 可以是一个表示目录的字符串或文件描述符。例如：

```
>>>os.chdir('E:\\hwdong_courses\\python')
```

os.**listdir**(path=None)：返回 path 目录下的所有文件(夹)的 list 对象。如果 path 是 None，则其值是当前目录，即'.'。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
>>>os.listdir('./imgs')
['binary_system.png', 'dirA', 'Hexadecimal.png', 'src2', 'text.jpg', 'text.png']
```

以下命令返回'./imgs'目录下的子目录：

```
>>>[x for x in os.listdir('./imgs') if os.path.isdir(x)]
['dirA', 'src2']
```

os.**mkdir**(path, mode=511, *, dirfd=None)：以数字权限模式 mode(对 window 系统无效)创建一个目录 path。如果 dirfd 不是 None，则它必须是一个目录的文件描述符，而 path 就是相对于这个目录的相对路径。当目标目录已经存在或中间目录不存在时，os.mkdir()会出错。例如，D 盘没有 hwdong 目录，执行下面代码将报错：

```
>>>os.mkdir('d:\\hwdong\\hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [WinError 3] 系统找不到指定的路径。: 'd:\\hwdong\\hello'
```

os.**makedirs**(name, mode=511, exist_ok=False)：类似 mkdir()，但该函数可递归地创建目录，创建包括叶子目录的所有中间目录。exist_ok 默认值是 False，表示叶子目录存在时将抛出OSError异常。当中间目录不存在时，os.makedirs()会创建这些中间目录。即使 D 盘没有 hwdong 目录，下面代码也能创建这些中间目录：


```
>>> os.makedirs("D:\\hwdong\\hello")
```

os.**rmdir**(path)：删除空目录 path，如果 path 不是空的目录，则删除失败。

os.**removedirs**(path)：递归地删除空的叶子目录和所有空中间目录。假如执行 os.removedirs(r'E:\a\b\c')，如果叶子目录 c 是空目录则删除该目录，然后对它的父目录 E:\a\b 重复这个删除过程，直到路径上的某个目录不是空的目录。实际上，os.removedirs()是从空叶子目录到父目录不断调用 os.rmdir()以删除空目录。

os.**remove**(path, *, dirfd=None)：删除指定路径的文件。如果指定的路径是一个目录，则抛出异常 OSError。dirfd 如果不是 None，则必须是一个目录的文件描述符(句柄)，此时，path 就是相对于 dir_fd 目录的相对路径。

os.path.**isfile**(path)：如果路径 path 是常规文件，则返回 True，否则返回 False。

os.path.**isdir**(path)：如果 path 是目录，则返回 True，否则返回 False。例如：

```
>>>os.path.isdir('E:\\hwdong_courses\\python\\test.py')
False
>>>os.path.isfile('E:\\hwdong_courses\\python\\test.py')
True
```

os.**rename**(src, dst)：将一个文件(文件夹)src 重命名为 dst。例如，将文件名 test.txt 改名为 pytest.txt：

```
os.rename("test.txt", "pytest.txt")
```

os.**startfile**(path)：打开一个关联某程序的文件。例如：

```
os.startfile(r'C:\Users\mike\Documents\labels.pdf')
```

用 pdf 关联的阅读程序打开 pdf 文件(labels.pdf)。

os.**walk**(top, topdown=True, onerror=None, followlinks=False)：返回一个目录树的生成器。 对于以 top 为根的目录树中的每个目录(包括 top 本身，但不包括.和..)，产生一个三元组：

```
dirpath, dirnames, filenames
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

其中，dirpath 是表示目录路径的字符串，dirnames 是 dirpath 中子目录的名称(不包括.和..)的列表，filenames 是 dirpath 中非目录文件的名称列表。

例如：

```
>>> path = r'E:\\hwdong_courses\\python\imgs'
>>> for root, dirs, files in os.walk(path):
    print(root)
```

输出：

```
E:\\hwdong_courses\\python\imgs
E:\\hwdong_courses\\python\imgs\dirA
E:\\hwdong_courses\\python\imgs\dirA\dirB
E:\\hwdong_courses\\python\imgs\src2
```

上述语句仅输出了每个三元组的 root 目录。可以用如下代码循环访问一个目录下的所有目录和文件：

```
for root, dirs, files in os.walk(path):
      print(root)
      for _dir in dirs:
         print(_dir)
      for _file in files:
         print(_file)
```


os.**stat**(file)：获得文件属性。例如：

```
>>>os.stat('d:\\hwdong\string.cpp')
os.stat_result(st_mode=33206, st_ino=562949953459610, st_dev=2057333047,
st_nlink=1, st_uid=0, st_gid=0, st_size=2979, st_atime=1555980604,
st_mtime=1517562484, st_ctime=1555980604)
```

os.**chmod**(file)：修改文件权限和时间戳。

os.**system**()：运行 shell 命令。例如：

```
>>>os.system('mkdir src')          #运行操作系统命令 mkdir 创建一个新的目录 src
>>>print(os.path.isdir('src') )    #判断一个路径是否为一个目录
True
```

os.**exit**()：终止当前进程。

os.**linesep**()：给出当前平台的行终止符。例如，Windows 系统使用'\r\n'，Linux 系统使用'\n'，而 Mac 系统使用'\r'。

问：如何删除一个非空的目录？

答：可以组合使用 walk()、remove()、rmdir()等函数，也可以用高层文件操作模块 shutil 的函数 rmtree()。例如：

```
import shutil
shutil.rmtree('/path/to/your/dir/')
```

2．os.path 子模块

os.path 子模块是 os 模块和路径相关的子模块。

os.path.**basename**：返回路径名的最后的文件或目录名。例如：

```
>>> os.path.basename(r'E:\\hwdong_courses\\python')
'python'
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

os.path.**dirname**(path)：返回文件的目录名。例如：

```
>>>os.path.dirname('E:\\hwdong_courses\\python\\test.py')
'E:\\hwdong_courses\\python'
```

os.path.**split**()：分离一个路径的目录名和文件名。例如：

```
>>> os.path.split('E:\\hwdong_courses\\python\imgs')
('E:\\hwdong_courses\\python', 'imgs')
```

os.path.**splitext**()：分离文件名和扩展名。例如：

```
>>>os.path.splitext('/Users/ethan/coding/python/hello.py')
('/Users/ethan/coding/python/hello', '.py')
```

os.path.**exists**(path)：如果 path 路径存在，则返回 True，否则返回 False。

os.path.**getsize**(path)：返回 os.stat 给出的 path 文件的大小(字节)。如果 path 是目录，则返回 0L。

os.path.**join**(path,name)：组合多个路径。第一个绝对路径前面的参数被忽略。例如：

```
>>> os.path.join('c:\\', 'csv', 'test.csv')
'c:\\csv\\test.csv'
>>> os.path.join('windows\temp', 'c:\\', 'csv', 'test.csv')
'c:\\csv\\test.csv'
```


os.path.**abspath**(name)：返回一个当前目录，并将其作为路径前缀的绝对路径。例如：

```
>>>os.path.abspath('test.py')
'E:\\hwdong_courses\\python\\test.py'
```

os.path.**isabs**()：判断是否为绝对路径。

os.path.**normcase**(path)：在 Linux 和 Mac 系统中，该函数会原样返回 path，在 Windows 系统中，该函数将路径中所有字符转换为小写，并将所有斜杠转换为反斜杠。例如：

```
>>> os.path.normcase('c:/windows/system32/')
'c:\\windows\\system32\\'
```

os.path.**normpath**(path)：通过折叠冗余分隔符和上级引用来规范化路径名，使得 A // B、A / B /、A /./ B 和 A / foo /../ B 等规范成同一个形式，如 A/B(Linux 或 Mac 系统)或 A\\B(Windows 系统)。

问：如何找出某一目录下的所有扩展名是.py 的 Python 程序文件？

答：可使用如下代码：

```
>>> [x for x in os.listdir('.') if os.path.isfile(x) and os.path.splitext(x)[1]== '.py']
```

输出：

```
['copytree.py', 'hello.py', 'hi.py', 'invaner.py', 'invasion.py',
'isolation.py', 'md2word-.py', 'md2word.py', 'mulprocess.py', 'myobj.py',
'pong_eng.py','pygame_pong.py','tcp_client_chunks.py','tcp_server_thread_notify.py',
'test.py', 'test2.py', 'test_slot.py', 'thread_enumerate.py', 'udp_client.py',
'udp_server.py', 'worker.py']
```

9.1.2 高层文件操作

shutil 模块提供了对文件和文件集合的许多高级操作，如文件复制和删除的功能。

1. 复制文件

shutil 模块主要有以下四个文件复制函数。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

(1) 复制函数 shutil.**copyfile**(src, dst)。

将源文件 src 复制到目标文件 dst。如果目标位置不可写，则抛出 IOError 异常。例如：

```
import shutil
src = 'text.txt'
dst = 'src.txt'
shutil.copyfile(src, dst)
```

(2) 复制函数 shutil.**copyfileobj**(src,dst[, length])。

copyfile()利用低层的 **copyfileobj**()实现复制，copyfileobj()的函数规范是：

```
shutil.copyfileobj(src, dst[, length])
```

copyfile()的参数是文件名，而 copyfileobj()的参数是文件句柄。此外，copyfileobj()还有第三个参数即缓冲区大小，当该值是正值时，将以该值大小逐块地从源文件读取数据并复制到目的文件中。

(3) 复制函数 shutil.**copy**(src, dst)。

该函数的工作方式与 copyfile()类似，区别是 copy(src, dst)的 dst 如果是一个目录，则会在这个目录下创建一个和源文件名一样的目标文件，而 copyfile()的 dst 则必须是一个文件。

(4) 复制函数 shutil.**copy2**(src, dst)。

该函数的工作方式与 copy()类似，但还会将元数据(如访问和修改时间)也复制到新文件中。

例如，可以用如下程序来验证元数据是否一起复制到目标文件中：

```
import os
import shutil
import time

def show_file_info(filename):
    stat_info = os.stat(filename)
    print('  Mode    :', oct(stat_info.st_mode))
    print('  Created :', time.ctime(stat_info.st_ctime))
    print('  Accessed:', time.ctime(stat_info.st_atime))
    print('  Modified:', time.ctime(stat_info.st_mtime))

os.mkdir('tmp')
print('SOURCE:')
show_file_info('test.py')

shutil.copy2('test.py', 'tmp')

print('DEST:')
show_file_info('tmp/test.py')
```

程序输出：

```
SOURCE:
  Mode    : 0o100666
  Created : Sun Jul  8 15:54:12 2018
  Accessed: Sun Jul  8 15:54:12 2018
  Modified: Sat Jun 16 08:12:56 2018
DEST:
  Mode    : 0o100666
  Created : Thu Jul 12 19:54:55 2018
  Accessed: Sun Jul  8 15:54:12 2018
  Modified: Sat Jun 16 08:12:56 2018
```

当然，文件复制也可以利用 os 模块的 system 调用方法，直接调用底层操作系统的文件复制(拷贝)命令。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import os
if os.name == 'nt':
   #Windows 的文件复制命令是 copy
   os.system('copy file1.txt file7.txt')
else:
   #Unix 的文件复制命令是 cp
   os.system('cp file1.txt file7.txt')
```

2．复制文件权限

默认情况下，在 Unix 系统中创建新文件时采用当前用户的 umask 权限。如果要将权限从一个文件复制到另一个文件，则可使用 copymode()。例如，下面的代码将一个文件的权限复制给另一个文件。

```
import os
import shutil
import subprocess

with open('text.txt', 'wt') as f:
    f.write('content')
os.chmod('text.txt', 0o444)

print('BEFORE:', oct(os.stat('text.txt').st_mode))
shutil.copymode('test.py', 'text.txt')
print('AFTER :', oct(os.stat('text.txt').st_mode))
```


程序输出：

```
BEFORE: 0o100444
AFTER : 0o100666
BEFORE: 0o100444
AFTER : 0o100666
```

如果在复制时，不仅要复制权限，而且还要复制其他元数据，如访问或修改时间等，则可以使用 shutil.**copystate**(src, dst)。

3．移动文件

shutil.move()可以将文件或目录移到另一个目录中。例如：

```
shutil.move(src, dst, copy_function=copy2)
```

该函数递归地将文件或目录从 src 移到 dst 中，其中，第三个参数指示采用哪个 copy 函数。

如果目标 dst 是已存在的目录，则 src 将移动到该目录中。如果目标已存在但不是目录，则可能根据 os.rename()语义覆盖它。

例如，下面的代码可将源目录 srcDir 中的内容移动到目标目录 dstDir 中：

```
import os, shutil
srcDir = "e:/srcdir"
dstDir = "e:/dstdir"
files = os.listdir(srcDir)
for f in files:
    src = srcDir+f
    dst = dstDir +f
    shutil.move(src,dst)
```

4．复制目录

shutil.copytree()用于将一个目录从一个地方复制(拷贝)到另外一个地方。其格式如下：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
shutil.copytree(src, dst, symlinks=False, ignore=None, copyfunction=copy2,
     ignoredangling_symlinks=False)
```

它递归地遍历 src 目录树，最终将文件复制到目的路径 dst。将被创建的目标目录（由 dst 命名）事前不可以存在。

参数 symlinks 用于控制将符号链接复制为链接还是复制为文件。默认设置（symlinks=False）是将内容复制到新文件，如果该选项为 true，则在目标树中创建新的符号链接。

两个可调用的参数（ignore 和 copyfunction）用于控制其行为。参数 ignore 是一个可调用对象，它接收一个目录，返回一系列相对于当前目录的目录和文件名，这些目录和文件名将在复制过程中被忽略。参数 copyfunction 是用于实际复制文件的可调用对象，默认为 copy2()。

下面的代码中，copyfunction 设置为自定义的叫作 verbose_copy 的文件复制函数。该函数只是在调用 copy2 复制文件前输出了一个打印语句。ignore_patterns()用于创建一个 ignore（忽略）函数，以跳过复制.png 后缀的文件：

```
import glob
import pprint
import shutil

def verbose_copy(src, dst):
    print('copying\n {!r}\n to {!r}'.format(src, dst))
    return shutil.copy2(src, dst)

print('BEFORE:')
pprint.pprint(glob.glob('./tmp/imgs/*'))

shutil.copytree(
    './imgs', './tmp/imgs',
    copy_function=verbose_copy,
    ignore=shutil.ignore_patterns('*.png'),
)

print('\nAFTER:')
pprint.pprint(glob.glob('./tmp/imgs/*'))
```

输出：

```
BEFORE:
[]
copying
 './imgs\\text.jpg'
 to './tmp/imgs\\text.jpg'

AFTER:
['./tmp/imgs\\dirA', './tmp/imgs\\src2', './tmp/imgs\\text.jpg']
```

5．删除目录

shutil.rmtree()用于删除整个目录树。其格式如下：

```
shutil.rmtree(path, ignore_errors=False, onerror=None)
```

参数 path 必须指向目录（但不是指向目录的符号链接）。如果参数 ignore_errors 为 True，则忽略由删除失败导致的错误，如果为 False 或省略，则通过调用 onerror 指定的处理程序来处理此类错误，此时如果 onerror 为 None，则会引发异常。例如：

```
import glob
import pprint
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import shutil

print('BEFORE:')
pprint.pprint(glob.glob('./tmp/imgs/*'))

shutil.rmtree('./tmp/imgs')

print('\nAFTER:')
pprint.pprint(glob.glob('./tmp/imgs/*'))
```

输出：

```
BEFORE:
['./tmp/imgs\\dirA', './tmp/imgs\\src2', './tmp/imgs\\text.jpg']

AFTER:
[]
```

6．查找文件

shutil.which()函数返回可执行命令 cmd 的路径。其格式如下：

```
shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)
```


shutil.which()函数扫描搜索路径 path 以查找 cmd 文件。典型的用例是在 shell 的环境变量 path 中定义的搜索路径上查找 cmd 文件。如果未找到与搜索参数匹配的文件，则返回 None。

参数 mode 是传递给 os.access()的权限掩码，默认情况下确定文件是否存在且是可执行的。参数 path 默认为 os.environ('PATH')，但可以是包含由 os.pathsep 分隔目录名的任何字符串。

- os.F_OK：作为 os.access()的 mode 参数，测试 path 是否存在。
- os.R_OK：包含在 os.access()的 mode 参数中，测试 path 是否可读。
- os.W_OK：包含在 os.access()的 mode 参数中，测试 path 是否可写。
- os.X_OK：包含在 os.access()的 mode 参数中，测试 path 是否可执行。

例如：

```
import shutil


print(shutil.which('virtualenv'))
print(shutil.which('python'))
print(shutil.which('no-such-program'))
```

输出：

```
None
E:\Anaconda\python.EXE
None
```

7．压缩文件、解压文件

查看支持的压缩文件格式，可用下面的函数：

```
shutil.get_archive_formats()
```

执行上述语句，可能的输出如下：

```
[('bztar', "bzip2'ed tar-file"),
 ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'),
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        ('xztar', "xz'ed tar-file"),
        ('zip', 'ZIP file')]
```

压缩文件用函数 shutil.make_archive()，其格式如下：

```
    shutil.make_archive(base_name, format[, root_dir])
```

其中，函数 root_dir 是要压缩的文件夹(目录)，参数 base_name 就是压缩后的文件名，参数 format 是压缩的格式，如 zip、tar 等。例如：

```
    shutil.make_archive('AAA', 'zip', 'imgs')  #压缩文件
```

输出：

```
    'E:\\hwdong_courses\\python\\AAA.zip'
```

解压文件用函数 shutil.unpack_archive()，其格式如下：

```
    shutil.unpack_archive(filename[, extract_dir[, format]])
```

其中，参数 extract_dir 是解压后的文件夹。例如，将新压缩的文件 AAA.zip 解压到 BBB 目录中：

```
    shutil.unpack_archive('AAA.zip','BBB')    #解压文件
```

8．获取磁盘使用空间

用函数 shutil.disk_usage(path)查看指定路径 path 的磁盘使用情况。例如：

```
    import shutil
    gb = 1024 ** 3   #GB == gigabyte
    total_b, used_b, free_b = shutil.disk_usage('E:')    #查看磁盘的使用情况
    print('总的磁盘空间: {:6.2f} GB '.format( total_b / gb))
    print('已经使用的 : {:6.2f} GB '.format( used_b / gb))
    print('未使用的 : {:6.2f} GB '.format( free_b / gb))
```

输出：

```
    总的磁盘空间: 114.06 GB
    已经使用的 : 110.02 GB
    未使用的 :  4.04 GB
```

9.1.3 glob 模块

英文“globbing”意为统配，glob 模块提供了根据 Unix Shell 通配符规则搜索匹配文件的函数，即根据通配符符号进行模糊搜索。例如，*可以匹配任意多个字符，?可以则匹配单个字符，中括号[]指定匹配的范围，如[0-9]匹配 1 至 9 范围内的数字。

glob 模块的两个主要函数功能是一样的，但是一个直接返回所有结果，另一个返回一个迭代器。

- glob(pathname, recursive=False)：参数 pathname 为需要匹配的字符串，参数 recursive 表示是否为递归调用，函数返回匹配的文件列表。
- iglob(pathname, recursive=False)：该函数的参数与函数 glob()的参数一致，但返回的是一个迭代器，当数据量非常大时，建议使用该函数。

例如，下面的代码搜索文件扩展名为“.py”的所有文件：

```
    import glob
    glob.glob('*.py')
```

下列代码搜索并打印目录 dir 下最后一个字符是数字的所有文件名：

```
    import glob
    for name in sorted(glob.glob('dir/*[0-9].*')):
       print(name)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

9.2 时间和日期模块

Python 的 time、datetime、calendar 模块提供了和时间、日期、日历相关的功能。下面介绍时间(time)模块和日期模块(datetime)。

9.2.1 时间模块

1. 相关术语

UTC(Coordinated Universal Time，协调世界时)，又称格林尼治天文时间、世界标准时间。与 UTC 对应的是各个时区的 local time(本地时间)，东 N 区的时间比 UTC 早 N 个小时，因此，UTC + N 为东 N 区的本地时间，而西 N 区时间比 UTC 晚 N 个小时，因此，UTC − N 为西 N 区的本地时间。中国在东 8 区，因此比 UTC 早 8 小时，可以用 UTC+8 表示。

epoch time(**纪元时间**)，表示一个特定的开始时间，不同平台上该时间点的值不太相同，在 Unix 系统中，epoch time 为 1970-01-01 00:00:00 UTC(1970 年 1 月 1 日 0 时 0 分 0 秒)。

timestamp(**时间戳**)，也称 Unix 时间或 POSIX 时间，它表示从 epoch time(如 1970-01-01 00:00:00 UTC)开始到现在所经过的毫秒数，其值为 float 类型。但是，有些编程语言的相关方法返回的是秒数(Python 就是这样)。时间戳是个差值，其值与时区无关。

2. time 模块

time 模块提供了不同的时间相关的函数。例如，time 模块的函数 time()返回的是 timestamp 值：

```
import time
print( "time.time(): %f " %  time.time())
```

以上输出为从纪元时间到当前时刻经过的秒数：

```
time.time(): 1538547221.304198
```

可以用函数 time.ctime([secs])将时间戳转为字符串(本地时间字符串)：

```
t = time.time()
print(t)
print( time.ctime( t ) )
```

以上输出为时间戳对应的本地时间字符串：

```
1538547913.0730906
Wed Oct  3 14:25:13 2018
```

可使用函数 time.localtime([secs])将时间戳转换为 **struct_time** 类型的本地时间：

```
st = time.localtime(t)
print(type(st))
print(st)
```

以上输出为 time.struct_time 类型的值：

```
<class 'time.struct_time'>
time.struct_time(tm_year=2018, tm_mon=10, tm_mday=3, tm_hour=14, tm_min=25,
tm_sec=13, tm_wday=2, tm_yday=276, tm_isdst=0)
```

time.struct_time 的属性名及下标如表 9-1 所示。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

表 9-1　time.struct_time 的属性名及下标

下标/索引	属性名称	描　　述
0	tm_year	年份，如 2017
1	tm_mon	月份，取值范围为[1,12]
2	tm_mday	一个月中的第几天，取值范围为[1,31]
3	tm_hour	小时，取值范围为[0,23]
4	tm_min	分钟，取值范围为[0,59]
5	tm_sec	秒，取值范围为[0,61]
6	tm_wday	一个星期中的第几天，取值范围为[0,6]，0 表示星期一
7	tm_yday	一年中的第几天，取值范围为[1,366]
8	tm_isdst	是否为夏令时，可取值为：0, 1 或−1

既可以通过下标(如 st[0])，也可以通过属性名(如 st.tm_year)访问 time.struct_time 对象的每个属性。例如：

```
print(st[0],'\t',st.tm_year)
```

输出：

```
2018 2018
```

还可以通过函数 time.asctime(t)将元组形式或 time.struct_time 类型对象的时间转换为一个字符串，如“Sun Jun 20 23:21:05 1993”。例如：

```
print( time.asctime( time.localtime(time.time()) ) )
```

输出：

```
Wed Oct  3 14:49:11 2018
```


9.2.2　日期模块

datetime 模块提供可用于处理日期和时间信息的函数和类，如对日期和时间解析、格式化和算术运算。

datetime.date：用于与时间无关的日期。

datetime.time：用于独立于日期的时间。

datetime.datetime：用于具有日期和时间的对象。

datetime.timedelta：表示日期或日期时间之间的差异，如果用一个日期时间减另一个日期时间，结果将是 timedelta。

datetime.timezone：表示时区调整为 UTC 的偏移量。该类是 datetime.tzinfo 的子类，不应直接使用。

可以查询这些对象的特定成分(如年、月、日，时、分、秒)，并对它们执行算术运算，如果需要显示它们，则可从中提取可以打印的字符串版本。

1．datetime.time 类

datetime.time 类是用于表示具有小时、分钟、秒、微秒和时区信息等属性的时间类(time 类)。例如：

```
import datetime

t = datetime.time(1, 2, 3,23)
print(t)
print(t.hour,t.minute,t.second,t.microsecond,t.tzinfo)
```

输出：

```
01:02:03.000023
1 2 3 23 None
```

time 类的 min 和 max 属性是一天中时间的最小和最大值。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import datetime
print('{:^12} {:^12} {:^12} '.format('最早','最迟','分辨率'))
print(datetime.time.min,'\t',datetime.time.max,'\t',datetime.time.resolution)
```

输出：

```
    最早                  最迟            分辨率
00:00:00      23:59:59.999999  0:00:00.000001
```

2．datetime.date 类

datetime.**date** 类是表示具有年、月、日属性的类。使用 today()类方法可返回当前日期的 date 对象。例如：

```
import datetime

today = datetime.date.today()
print(today)
print(today.year, today.month, today.day)
```

输出：

```
2018-10-03
2018 10 3
```


可以利用 date 类的 ctime()方法返回一个日期的字符串表示。例如：

```
print('ctime  :', today.ctime())
```

输出：

```
ctime  : Wed Oct  3 00:00:00 2018
```

可以使用 date 类的 timetuple ()方法返回一个 time.struct_time 对象。例如：

```
tt = today.timetuple()
print(tt)
print('{:^9} {:^9} {:^9} {:^9} {:^9} {:^9}{:^9} {:^9} {:^9}'.format(
    'tm_year','tm_mon','tm_mday','tm_hour','tm_min','tm_sec','tm_wday','tm_yday','tm_isdst'))
print('{:^9} {:^9} {:^9} {:^9} {:^9} {:^9}{:^9} {:^9} {:^9}'.format(
tt.tm_year,tt.tm_mon,tt.tm_mday,tt.tm_hour,tt.tm_min,tt.tm_sec,tt.tm_wday,
tt.tm_yday,tt.tm_isdst))
```

输出：

```
time.struct_time(tm_year=2018, tm_mon=10, tm_mday=3, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=2, tm_yday=276, tm_isdst=-1)
 tm_year   tm_mon    tm_mday   tm_hour   tm_min    tm_sec   tm_wday   tm_yday  tm_isdst
  2018      10        3         0         0         0        2         276       -1
```

与 time 类一样，可以使用 min 和 max 属性确定支持的日期值范围。同样，date 的 resolution（分辨率）是整数的天。

```
print('{:^9} {:^20} {:^30} '.format('最早','最迟','分辨率'))
print(datetime.date.min,'\t',datetime.date.max,'\t',datetime.date.resolution)
```

输出：

```
   最早                 最迟                 分辨率
0001-01-01    9999-12-31          1 day, 0:00:00
```

通过构造函数或 replace()方法可以创建一个新的日期实例。例如：

```
d1 = datetime.date(2017, 1, 29)
print('d1:', d1.ctime())
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
d2 = d1.replace(year=2018)
print('d2:', d2.ctime())
```

输出：

```
d1: Sun Jan 29 00:00:00 2017
d2: Mon Jan 29 00:00:00 2018
```

datetime.date 类的 fromtimestamp() 方法可以将一个时间戳转换为一个 date 类对象。例如：

```
import time
import datetime
today = datetime.date.fromtimestamp(time.time())
print(today)
```

输出：

```
2018-10-03
```

datetime.datetime 类表示时间和日期的组合，可以用 combine() 函数组合日期和时间，也可用 datetime() 函数构造一个包含日期和时间的 datetime 对象。例如：

```
import datetime
t = datetime.time(1, 2, 3)
d = datetime.date.today()
now = datetime.datetime.combine(d, t)
long_ago = datetime.datetime(1999, 3, 14, 12, 30, 58)
print(now)
print(long_ago)
print(now.strftime("%a, %d %B %Y"))    #使用 strftime()将 ISO-8601 格式
                                       #(YYYY-MM-DD HH:MM:SS:mmmmmm)
                                       #转换为定制格式的字符串表示
```

输出：

```
2018-10-03 01:02:03
1999-03-14 12:30:58
Wed, 03 October 2018
```

strftime() 函数的格式可以在以下网址查看：

```
https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior
```

datetime.datetime 类的 fromtimestamp() 方法可以将一个时间戳转换为一个 datetime 类对象。例如：

```
today = datetime.datetime.fromtimestamp(time.time())
print(today)
```

输出：

```
2018-10-03 17:30:47.748808
```

timedelta(时间差)对象表示一段时间间隔。例如，两个时间对象相减的结果就是一个 timedelta 对象，而一个时间对象加上一个 timedelta 对象就会产生一个新的时间对象。

例如，两个 date 相减可以产生一个 timedelta(时间差)对象，同样，一个 date 可以加或减一个 timedelta，就可以得到一个新的 date。

timedelta 的内部值以天、秒和微秒存储。例如：

```
difference = now - long_ago   #两个 date 相减可以产生一个 timedelta(时间差)对象
print(type(difference))
print(difference)
print(difference.days,difference.seconds,difference.microseconds)
t3 = datetime.timedelta(days=3,hours=9,minutes=45)      #构造一个 timedelta 对象
day_3 = now +  t3
print('day_3 :', day_3)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
<class 'datetime.timedelta'>
7142 days, 12:31:05
7142 45065 0
day_3 : 2018-10-06 10:47:03
```

9.3 习题

1. 编写一个函数 findfiles()递归地遍历一个目录中的所有子目录和文件，并输出这些子目录和文件的路径。
2. 编写一个函数，输出一个目录中扩展名为“.txt”的所有文件名。
3. 编写一个函数，计算一个目录中所有后缀是“.py”的文件中代码行的个数(忽略空行和注释行)。
4. 下列代码实现文件复制功能，输入源文件所在路径和目标目录路径，将源文件复制到目标目录中。在此基础上，实现将一个目录中所有文件复制到目标路径的功能。注意，不能用 shutil 模块中的函数。

```
#coding=utf-8
import os

file_source = input('请输入要复制的文件的绝对路径:')
file_dest = input('请输入目标路径:')

#调用 os.path 模块函数，将输入的文件路径规范化
list_name = os.path.normcase(file_source)
list_dir = os.path.normcase(file_dest)
#源文件存在则复制，否则提示复制失败
if os.path.exists(list_name) :
  #目标路径如果是存在的目录则复制，否则递归创建目标目录
  if os.path.isdir(list_dir) :
    #拼接 dos 下执行的复制文件的命令
    nstr = 'copy' + ' ' + list_name + ' ' + list_dir
  else :
    os.makedirs(list_dir)
    nstr = 'copy' + ' ' + list_name + ' ' + list_dir
else :
  print ("复制的文件不存在！")
  exit()
print("开始复制：",nstr)
#调用 os 模块函数，执行 dos 下的复制命令，并返回结果
flag = os.system(nstr)
if flag == 0 :
  print("文件复制成功！")
else :
  print( "文件复制失败！")
```

5. 构造一个 datetime 对象，从程序运行的当前时间倒退 8 小时 30 份 50 秒，然后将结果使用函数 strftime()以年、月、日、时、分、秒格式输出，并将该 datetime 对象转化为时间戳并输出。
6. 编写一个程序，将一个字符串转为 datetime 对象。假如输入“Jan 1 2014 2:43PM”则输出“2014-07-01 14:43:00”。注意：可使用函数 datetime.strptime()完成转换工作。
7. 编写一个程序，输出两个日期(date)之间的所有日期(date)。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 10 章　正则表达式

10.1　正则表达式的定义

计算机处理的数据主要包括文本、图像、语音，其中文本数据的处理最多，应用也最广泛。文本就是一系列字符(也称“**字符串**”)，在处理文本的过程中，字符串搜索功能，或者说“**模式匹配**”，是最重要的字符串操作，其含义就是在一个字符串中搜索满足特定条件或规则的字符串子序列(也称“**子串**”)。例如，在一篇文章中寻找某个特定的文本(字符串)如“Python”，被搜索的文章也是一个字符串，称为“**主串**”，而用来刻画搜索条件或规则的字符串如“Python”称为“**模式串**”。再如，要在字符串“hi，how are you”中搜索“how”，那么“how”就是“模式串”，而“hi，how are you”就是主串。

每种程序语言中都有字符串这种数据类型，并且程序针对字符串类型都提供了模式匹配功能，即字符串搜索功能。Python 的字符串类型 str 的 find()方法提供了模式匹配功能，对于上面的例子，可以用下列代码来执行字符串查找：

```
print("hi,how are you".find('how'))
3
```

这种逐字符地模式匹配有很大局限性。例如，要求在一个文本中查找所有以 c 开头并以 t 结尾的字符串，这样的字符串有很多个，如“cat”“cut”“cheat”等，但无法用简单的逐字符精确匹配的方法完成这种任务。这样的例子很多，如在一个文章或网页中提取所有 email 地址，网络爬虫提取网页中的所有图片地址，检查程序中的标识符是否合法，利用搜索引擎搜索满足条件的网页文章等。

针对这些复杂的模式匹配，一种解决方法是对每个具体任务编写一个专门的模块或函数，如编写一个专门从文章中提取 email 地址的函数。但实际的任务可以无穷多个，为每个这样的任务编写专门的模块不仅费时费力，也使程序代码越来越多，出错率会越来越大，维护软件也变得很困难。

有没有一种通用的一劳永逸的方法呢？答案是：有，使用强大灵活的“**正则表达式**”！

正则表达式是用普通字符和称为“元字符”的特殊字符构成的一个字符串。它用于描述匹配某种规则的所有字符串的特征。

前面的“how”就是一个正则表达式，其中只有三个普通字符，如果一个字符串的子串和“how”相等就表示匹配。这种普通字符只能表示逐字符的匹配，而具有特殊含义的“元字符”可以表达更加复杂的匹配规则。

例如，元字符 . 表示可以匹配除换行符\n 之外的任何单个字符。因此，由两个普通字符 r、t 中间夹一个元字符 . 组成的正则表达式 r.t 表示“匹配一个 r 接着任意一个非换行字符再接着一个 t”这一匹配规则。例如，用这个正则表达式匹配下面的文本：

```
he is a rat
he is in a rut
the food is Rotten
I like root beer
```

则会匹配其中的子串“rat”和“rut”，但不会匹配“Rot”和“root”。

再如，元字符+表示其左边的模式可以出现 1 次以上，如 x+表示模式 x 可以出现 1 次、2 次、…，

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

即模式 x 出现 1 次以上。而元字符*表示其左边的模式可以出现 0 次、1 次、…，即表示既可以不出现，也可以出现 1 次以上。而元字符?表示其左边的模式 x 只能出现 0 次或 1 次，即最多出现 1 次。

方括号[]这种元字符用于描述一个字符集，匹配时只能匹配其中的一个字符。例如，[abc]表示匹配的字符只能是 a、b 或 c。[0123456789]表示只匹配 0 到 9 这 10 个数字字符中的 1 个。[]中可以用连字符-表示一个起始字符到结束字符范围里的所有字符。正则表达式[0123456789]可以简化为[0-9]。类似地，[a-z]表示所有小写字母，[a-zA-Z]表示所有大小写字母，[0-9a-zA-Z]表示所有字母和数字字符。

正则表达式中的元字符反斜杠 \ 可以和其他字符结合起来表示某种特殊含义。例如，\d 表示所有数字字符，即\d 等价于[0-9]；\w 表示大小写字母和数字字符，即\w 等价于[0-9a-zA-Z_]；\s 表示匹配 1 个空白字符。空白字符包括空格、换行、回车、制表、换页、纵向制表，即\s 等价于[\t\n\r\f\v]。

因此，正则表达式\s*表示 0 个或多个空白字符。而\d\s*\d\s*\d 表示被“0 个或多个空白字符”隔开的 3 个数字字符。例如，用模式串搜索下列字符串：

```
'xx1 2   3xx'
'xx12  3xx'
'xx123xx'
```

则分别产生如下的匹配结果：

```
'1 2   3'
'12  3'
'123'
```


其中，第 1 个结果是 1 个或多个空白字符分隔的，而最后 1 个结果是由“0 个空白字符”分隔的，即这 3 个数字字符之间没有任何空白字符。

10.2 re 模块

许多编程语言都会提供正则表达式的功能，即提供“根据正则表达式，在一个字符串中搜索满足条件的匹配正则表达式的子串”的功能。Python 语言自带的 re 模块提供了正则表达式的功能。只要导入：

```
import re
```

就可以直接用 re 模块的函数 search()，在字符串 str 对象(string)中搜索满足模式串(pattern)的子串。例如：

```
match = re.search(pattern, string)
```

该函数返回一个 match 对象，通过这个对象调用 match 类的 group()方法，则可以得到匹配的子串。例如：

```
import re
pattern = '\d\s*\d\s*\d'
match = re.search(pattern, 'xx1 2   3xx')
print(match.group())
match = re.search(pattern, "xx12  3xx")
print(match.group())
match = re.search(pattern, 'xx123xx')
print(match.group())
```

输出：

```
1 2   3
12  3
123
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

再如，用模式串 r.t 搜索字符串(如“he is Arat”)中匹配它的子串：

```
import re
pattern = 'r.t'
slist = [ "he is Arat","the food is rotten"]
for s in slist:
   match = re.search(pattern, s)
   if match:    print('找到匹配的子串：','\t',match.group())
   else:   print('未找到匹配模式串"{}"的子串'.format(pattern))
```

输出：

```
找到匹配的子串：  rat
未找到匹配模式串"r.t"的子串
```

可以看到，第一个字符串有一个匹配的子串“rat”，而第二个字符串未找到匹配的子串。通过 match 对象的 span()方法可以得到匹配的子串在主串中的起始位置和结束位置：

```
match = re.search("r.t", "he is Arat")
match.span()
```

输出：

```
(7, 10)
```

直接打印 match 对象，就可以看到这些信息(子串在主串中的起始位置、结束位置、子串内容)：

```
print(match)
```

输出：

```
<_sre.SRE_Match object; span=(7, 10), match='rat'>
```


10.2.1 re 模块的常用函数

re 模块提供了不同的函数用于执行不同的模式匹配功能，常用的函数如下：

```
re.match()
re.search()
re.findall()
re.finditer()
re.split()
re.sub()
re.compile()
```

10.2.2 编译模式串

除函数 re.compile()外，其他函数都执行不同的字符串搜索功能。如果要多次使用同一个正则表达式，为了提高执行效率，则可以使用函数 re.compile()将这个正则表达式首先编译为一个正则表达式对象(Pattern 类的对象)，然后通过 Pattern 类的 re 模块的不同搜索方法搜索一个字符串中符合这个正则表达式的子串。

例如，上面的正则表达式“r.t”可以首先使用函数 re.compile()编译为正则表达式对象，再通过这个对象搜索一个字符串：

```
import re
pattern = 'r.t'
p = re.compile(pattern)          #将正则表达式编译为正则表达式对象
print(type(p))

slist = [ "he is arat","the food is Rotten"]
for s in slist:
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
match = p.search(s)          #通过正则表达式对象 p，在字符串中搜索匹配它的子串
if match:    print(match.group())
else:   print('未找到匹配模式串"{}"的子串'.format(pattern))
```

上述代码首先将模式 pattern 编译为 Pattern 类对象 p，再通过函数 p.search()匹配主串 s。程序输出如下：

```
<class '_sre.SRE_Pattern'>
rat
未找到匹配模式串"r.t"的子串。
```

10.2.3 从头匹配

re 模块的函数 search()在一个主串中搜索和模式串匹配的第一个子串，如果成功，则返回一个 match 对象，否则，返回 None。

函数 match()用字符串开头的零个或多个字符构成的子串与正则表达式模式匹配，如果成功，则返回一个 match 对象，否则，返回 None。match()函数无须搜索就可以直接从字符串开头检查是否和模式串匹配。例如：

```
import re
text = 'the cat is there,there is a cat'
print(re.match('the', text))
print( re.match('there', text))
print(re.search('the', text))
print( re.search('there', text))
```


输出：

```
<_sre.SRE_Match object; span=(0, 3), match='the'>
None
<_sre.SRE_Match object; span=(0, 3), match='the'>
<_sre.SRE_Match object; span=(11, 16), match='there'>
```

函数 match()和函数 search()的区别是：函数 match()仅在字符串的开头检查匹配(默认情况下)，而函数 search()检查字符串中任意位置的匹配子串。

10.2.4 多个匹配

1．函数 findall()

re 模块的函数 findall()返回主串中所有和模式串匹配的(不重叠的)子串构成的 list 对象。

可以通过 for 循环遍历其中每个匹配的子串或分组。例如：

```
import re
ret = re.findall('ab', 'abbaaabbbbaaaaa')
print(ret)
```

输出：

```
['ab', 'ab']
```

即找到了两个和模式“ab”匹配的子串“ab”。

2．函数 finditer()

re 模块的函数 findall()一次性返回所有匹配结果，而函数 finditer()返回一个迭代器，遍历迭代器可以得到每个匹配的 match 分组。

```
import re
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
for match in re.finditer('ab', 'abbaaabbbbaaaaa'):
    s = match.start()
    e = match.end()
    print('Found {!r} at {:d}:{:d}'.format(
        text[s:e], s, e))
```

输出：

```
<_sre.SRE_Match object; span=(0, 2), match='ab'>
Found 'ab' at 0:2
<_sre.SRE_Match object; span=(5, 7), match='ab'>
Found 'ab' at 5:7
```

函数 finditer()、函数 findall()和函数 search()的区别如下。

- 函数 search()查找字符串中和模式串第一次匹配的子串。
- 函数 findall()、函数 finditer()查找字符串中和模式串匹配的所有不重叠的子串。

10.2.5 按匹配切分

字符串类型 str 的 split()方法支持用分隔符拆分一个字符串。例如：

```
s="aa bb  cc"
print(s.split(' '))        #用空格拆分字符串，返回拆分后的字符串的 list 列表对象
s="aa,bb, cc"
print(s.split(','))        #用逗号拆分字符串，返回拆分后的字符串的 list 列表对象
```

输出：

```
['aa', 'bb', '', 'cc']
['aa', 'bb', ' cc']
```

但 str 的 split()方法不支持多个分隔符，也无法支持更复杂的正则表达式表示的分隔模式。例如，无法分割用不同分隔符分隔的字符串：

```
s="aa.bb;cc,dd"
```

re 模块的 split()函数可以用指定模式来拆分字符串。其函数规范是：

```
re.split(pattern, string, maxsplit=0, flags=0)
```

第一个参数 pattern 是用作分隔模式的正则表达式；第二个参数是输入的字符串；第三个可选参数 maxsplit 如果非零，则最多发生 maxsplit 个拆分，并且字符串的其余部分将作为列表的最后一个元素返回。例如：

```
import re
s="aa.bb;cc,dd"
s1="aa,bb;cc"
s2 = 'aa bb.;cc'
s3 = 'aa;bb;cc'
print(re.split('[ .,;]',s))            #支持多个分隔符
print(re.split('[ .,;]',s1))
print(re.split('[ .,;]',s2))
print(re.split('[ .,;]',s3))
```

输出：

```
['aa', 'bb', 'cc', 'dd']
['aa', 'bb', 'cc', 'dd']
['aa', 'bb', '', 'cc']
['aa', 'bb', 'cc']
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

可以看到，一个正则表达式[.,;]可以用于分隔不同分隔符表示的字符串。

例如，下列代码将一段文本的每一行分隔出来：

```
text = """王文
          赵丹
          li ping
          Zhang wei """
results = re.split(r"\n+", text)
print(results)
```

输出：

```
['王文', '          赵丹', '          li ping', '          Zhang wei ']
```

10.2.6 替换匹配

字符串 str 的 replace()方法可以进行简单的字符串替换操作。例如：

```
text = 'yeah, but no, but yeah, but no, but yeah'
text.replace('yeah', 'yep')
```

输出：

```
'yep, but no, but yep, but no, but yep'
```


但是，当将下列用/连接的年、月、日字符串换成用–连接的年、月、日字符串时，则如果利用 str 的 repalce()方法就会产生错误的结果：

```
text = 'The dates of today/nextday are 07/21/2018 and  07/22/2018'
text.replace('/', '-')
```

输出：

```
'The dates of today-nextday are 07-21-2018 and  07-22-2018'
```

可以看到，非日期的子串 today/nextday 中的/也被替换为-了。

对于这类复杂(考虑上下文)字符串的替换操作，就要借助 re 模块的函数 sub()，其函数规范是：

```
replacedString = re.sub(pattern, replacement_pattern, string, count=0, flags=0)
```

第一个参数 pattern 是匹配模式的正则表达式；第二个参数 replacement_pattern 是要替换模式的正则表达式；第三个参数 string 是输入的文本；可选参数 count 是要替换的模式的最大出现次数，且 count 必须是非负整数。例如：

```
import re
text = 'The dates of today/nextday are 07/21/2018 and  07/22/2018'
re.sub('(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
```

即将和模式串(\d+)/(\d+)/(\d+)匹配的子串替换成符合模式串 r'\3-\1-\2'的子串。

输出：

```
'The dates of today/nextday are 2018-07-21 and  2018-07-22'
```

其中，正则表达式中的用( )包围的每个子表达式称为“**分组**”，每个分组都会自动分配一个编号，从左到右出现的分组编号依次为 1、2、3、...。正则表达式中可以用反斜杠和分组编号来引用前面的一个分组，以避免重复书写这些子表达式。

10.3 正则表达式中的语法规则

正则表达式是一个由普通字符和特殊的元字符构成的字符串。元字符可以表达各种复杂的规则。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

10.3.1 字符集

[]是一种表示字符集的元字符，字符集中虽然有多个字符，但只能匹配其中的一个字符。例如，[abc]表示匹配 a、b 或 c 中的任何一个；[0123456789]表示匹配这 10 个数字中 0、1、2、...、9 中的任意一个数字。

可以用连字符-来表示一个连续的字符范围。例如，[0123456789]可以用[0-9]表示；[a-z]表示匹配小写字母 a 到 z 的任意一个字符；但[a\-z]表示匹配三个字符 a、-、z 中的任意一个字符。因为这里的\-是转义字符表示普通字符-而不是元字符-；另外，[a-]和[-a]都表示的是两个字符 a 和-，因为作为连字符需要一个起始和结束；[a-z0-9]表示小写字母 a 到 z 和 0 到 9 中的任意一个。

一般的元字符如果在[]中，则通常就是普通字符类，如[(+*)]表示的是四个普通字符(、+、*、)。

在[]出现的元字符^表示“非”(排除)的意思，如[^abc]表示匹配 a、b、c 之外的任意一个字符。

除[]外，另有一些元字符和前面的转义字符等也表示不同字符集。例如，.匹配除换行符\n 外的任意一个字符，等价于[^\n]。

10.3.2 反斜杠

反斜杠\用于表示转义字符，主要有以下三个作用。

(1)表示不方便显示的特殊字符，如换行、制表符(Tab)等。此时，反斜杠后面可以跟一个字符或要表示的特殊字符对应的整数(十六进制、八进制数等)。表示特殊字符的转义字符及其含义如表 10-1 所示。

表 10-1 表示特殊字符的转义字符及其含义

转义字符	含 义
\n	表示换行符
\r	表示回车符
\t	表示制表符
\v	表示垂直制表符
\f	表示换页符
\x20	表示两位 16 进制数 x20 对应的 ASCII 字符
\u000A	表示 unicode 编码\u000A 对应的字符，实际就是换行符\n
\\	表示匹配反斜杠符\自身

(2)定义一些代表特殊意义的元字符。表示字符集或特殊意义的转义字符及其含义如表 10-2 所示。

表 10-2 表示字符集或特殊意义的转义字符及其含义

转义字符	含 义
\w	等价于 [0-9a-zA-Z_]：字母、数字或下画线
\W	等价于 [^0-9a-zA-Z_]：字母、数字或下画线以外的字符
\s	表示空白字符，等价于[\r\n\t\f\v]
\S	表示非空白字符, 等价于[^\r\n\t\f\v]
\d	表示数字字符，等价于[0-9]
\D	表示非数字字符，等价于[^0-9]
\b	表示匹配单词的边界。\b 代表单词的开头或结尾的分界处。但\b 并不匹配这些单词分隔字符中的任何一个，它只匹配一个位置。例如，要查找一个单词 he，可以使用\bhe\b 表示 he 的前后是分界
\B	表示匹配非单词边界

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

(3)表示元字符本身。元字符，如“^”表示特殊含义的匹配规则，那么如何像普通字符一样去匹配文本中的这种元字符呢？办法就是在元字符前加一个反斜杠，即\^表示普通字符^而不是元字符。

注意

因为 Python 语言也用反斜杠来定义转义字符，为了避免和正则表达式中的转义字符产生冲突，最好用 Python 的原始字符串的定义方式来定义正则表达式，即在表示正则表达式的字符串前添加字母 r。例如，r"\br.t\b"，即表示正则表达式中的\和 b 都是单独的字符，这两个字符合起来表示一个元字符\b。如果不这样做，那么\b 就会被看作转义字符，即一个字符；如果不使用原始字符串，则必须写为\\br.t\\b。

10.3.3 量词(重复)

如果希望某个字符或模式多次出现，则可在模式后面添加一个量词，用于说明该模式重复多少次，主要量词包括*、+、?、{}。量词及其含义如表 10-3 所示。

表 10-3 量词及其含义

量　词	含　义
{}	表示模式重复的次数。例如，{n}表示该模式重复出现 n 次，{m,n}表示该模式重复出现 m 到 n 次，优先匹配 n 次。例如，a{1,3}，可以表示 a 出现 1 到 3 次，可以匹配 aaa、aa、a。{m,}表示出现 m 到∞次，优先匹配∞次，即尽可能多地匹配，如 a{1,}可以匹配 aaaa...
?	表示模式出现 0 次或 1 次，优先匹配 1 次，相当于{0,1}
+	表示模式出现 1 次以上，优先匹配最大次数，相当于{1,}
*	表示模式出现 0 次以上，优先匹配最大次数，相当于{0,}。正则表达式 ab*表示“a 后面跟 0 或多个 b”

正则表达式默认采用贪婪模式，即尽可能匹配最长的字符串。凡是表示范围的量词，都优先匹配上限而不是下限。例如，用 a{1, 3}优先匹配 aaa 而不是 a。

10.3.4 边界字符(锚点)

有时希望正则表达式可以表示如以 xxx 开头或以 xxx 结尾的规则。例如，程序中的标识符必须以字母或下画线开头。

可以利用**边界字符(锚点)**描述模式串匹配主串中的某一位置的字符(子串)，如前面的\b 表示单词的边界。

锚点及其含义如表 10-4 所示。

表 10-4 锚点及其含义

编　码	含　义
^	串或行的开头
$	串或行的结尾
\A	串的开头
\Z	串的结尾
\b	单词开头或结束的分界位置，单词和空白字符的分界
\B	等价于：\B=[^\b]/ 单词之间的分界或空白字符之间的分界

元字符^表示匹配串或行的开头，$表示匹配串或行的结尾。元字符\A 表示匹配串的开头，\Z 表示匹配串的结尾。

例如，^he 表示字符串或行必须以 he 开头，而 ing$表示字符串或行必须以 ing 结尾：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
match = re.search(r'^he','he is sing')
print(match)
match = re.search(r'^he','she is dancing!')
print(match)
match = re.search(r'ing$','he is sing')
print(match)
match = re.search(r'ing$','she is dancing!')
print(match)
```

输出结果：

```
<_sre.SRE_Match object; span=(0, 2), match='he'>
None
<_sre.SRE_Match object; span=(7, 10), match='ing'>
None
```

再如，一个网站要求用户填写的 QQ 号必须为 5 位到 12 位数字，可结合量词和边界元字符，如使用正则表达式^\d{5,12}$来检验 QQ 号是否符合这个规则：

```
print(re.search(r'^\d{5,12}$','58486878'))
print(re.search(r'^\d{5,12}$','a58486878'))
print(re.search(r'^\d{5,12}$','a58486878x'))
print(re.search(r'^\d{5,12}$','1234567891123'))
```


可以发现，只有第一个是匹配的，其他的都不匹配，因此输出结果：

```
<_sre.SRE_Match object; span=(0, 8), match='58486878'>
None
None
None
```

10.3.5 或运算

当匹配 x 或 y 时，如果 x 和 y 是单个字符，则可使用字符集元字符[]，如[abc]可以匹配 a、b 或 c，如果 x 和 y 是多个字符，那么字符集就无能为力了，则需要用到将两个匹配条件进行逻辑“或”运算的元字符|。例如，123|456|789 表示匹配 123、456 或 789。正则表达式 him|her 可匹配“it belongs to him”和“it belongs to her”，但是不能匹配“it belongs to them.”。

10.3.6 分组

正则表达式中可以用圆括号()标记一个子表达式的开始和结束位置。一个子表达式也称为一个**分组**(group)。

正则表达式(123|456){2}表示匹配两次子表达式(123|456)，因此，字符串 123123、456456、123456、456123 都是可以匹配它的。

对于ip地址，如192.128.2.131可以用正则表达式(\d{1,3}.){3}\d{1,3}来检验。其中，分组(\d{1,3}.)表示匹配1位到3位的数字后面跟一个英文句号，(\d{1,3}.){3}表示匹配3个(\d{1,3}.)，最后的\d{1,3}再匹配1位到3位的数字。

1．分组的编号和引用

正则表达式中用()定义的每个分组都会自动分配一个编号，从左到右出现的分组的编号依次为1、2、3、...。正则表达式中可以用反斜杠和分组编号来引用前面定义的分组。

例如，<([a-z]+)><\/\1>的第一个分组是([a-z]+)，表示至少有一个以上的a到z的字母构成的字符串，

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

这个分组自动有一个编号 1。转义字符\/表示字符/。最后的\1 表示引用的是这个 1 号分组。因此，这个正则表达式可以匹配 html 或 xml 的首尾标签，即可以匹配字符串或 等。

正则表达式的“分组”功能允许选择匹配文本的一部分。假设分别提取电子邮件中的用户名和邮件服务器名，可以用两个分组表达式分别匹配用户名和服务器名，如 r'([\w.-] +)@([\w.-] +)'。在这种情况下，括号不会更改模式匹配的内容，而是在匹配文本内部建立逻辑“组”。在成功搜索时，match.group(1)是对应第一个分组的匹配文本，match.group(2)是对应第二个分组的匹配文本。普通的 match.group()仍然像往常一样是完整的匹配文本。例如：

```
str = 'This is the mail-box (教小白精通编程): xue.pro-8@gmail.com, please mail to us.'
match = re.search(r'([\w.-]+)@([\w.-]+)', str)
if match:
   print(match.group())   ##'xue.pro-8@gmail.com' (整个匹配字符串)
   print(match.group(1))  ##'xue.pro-8' (用户名 e, group 1)
   print( match.group(2)) ##'gmail.com' (主机, group 2)
```

输出：

```
xue.pro-8@gmail.com
xue.pro-8
gmail.com
```


2．扩展符号

分组里的?作为扩展符号，它的意义取决于它右边的字符。例如：

(?PAB)：匹配表达式 AB，可以通过组名访问它。

(?:A)：匹配由 A 表示的表达式，但与(?PAB)不同，不能通过组名访问它。

(?aiLmsux)：a、i、L、m、s、u 和 x 都是标志(这些标志可参考 10.4.2 节)：

a 只匹配 ASCII 字符；

i 忽略大小写；

L 依赖语言环境；

m 多行；

s 匹配所有字符；

u 匹配 unicode；

x 允许正则表达式可以是多行，忽略空白字符，并可以加入注释。

(?#...)：表示是一个注释，内容仅供阅读，而不是匹配。

A(?=B)：前瞻断言，仅当表达式 A 后跟 B 时才匹配表达式 A。

A(?!B)：否定前瞻断言，仅当表达式 A 后面没有 B 时，它才匹配表达式 A。

(?<=B)A：肯定回顾断言，仅当 B 紧靠其左侧时，才匹配表达式 A。它只能匹配固定长度的表达式。

(?<!B)A：否定回顾断言，仅当 B 不在其左侧时，才匹配表达式 A。它只能匹配固定长度的表达式。

(?P=name)：匹配已被匹配过的名为 name 的分组表达式。

例如：

```
import re
m = re.search('(?<=abc)def', 'abcdef')
print(m)
m = re.search('(?<=abc)def', 'abdef')
print(m)
```

输出：

```
<_sre.SRE_Match object; span=(3, 6), match='def'>
None
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

10.4 match 和 flags

10.4.1 match 对象及其应用

1. match 对象

match()和 search()如果查找失败，则返回一个 None 对象，如果查找成功，则返回一个 match 对象，并且当这个对象用于条件表达式时，其值可自动转换为 True，因此，可以用 if 语句判断查找是否成功。例如：

```
match = re.search(pattern, string)
if match:
   process(match)
```

match 的 group()方法返回匹配的一个或多个子组。其格式为：

```
match.group([group1,group2,...])
```

因为一个模式中可能有多个分组，所以通过 match.group([group1,group2,...])可以查询那些模式分组在被查询字符串中匹配的对应的子组(subgroups)。

可选参数[group1,group2,...]表示子组号，如果只有一个参数(子组号或子组名)，则返回该参数(子组号或子组名)对应的子组(单个字符串)，如果有多个参数，则返回的是多个子组构成的元组。如果 groupN 参数是 0，则返回结果是完整的匹配字符串。例如：

```
str = 'This is the mail-box: xue.pro-8@gmail.com, please mail to us.'
m = re.search(r'([\w.-]+)@([\w.-]+)', str)
if m:
   print(m.group())       #'xue.pro-8@gmail.com' (整个匹配字符串)
   print(m.group(1))      #'xue.pro-8' (用户名, group 1)
   print( m.group(2))     #'gmail.com' (主机, group 2)
   print(m.group(1, 2) )  #多个参数返回一个 tuple，每个元素是一个 subgroup(子组)
print(m.group(0))
```

上述代码的模式串中有两个分组，可通过返回的 match 对象 m 查询匹配的子组，如 m.group(1, 2)返回了两个匹配子组的元组。输出结果为：

```
xue.pro-8@gmail.com
xue.pro-8
gmail.com
('xue.pro-8', 'gmail.com')
xue.pro-8@gmail.com
```

正则表达式 r'([\w.-]+)@([\w.-]+)'定义了两个可捕获的分组，可以用分组号引用这两个分组。

如果正则表达式使用(? P ...)语法对分组进行命名，则 groupN 参数也可以是分组名。例如：

```
>>>m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

当然，命名分组仍然可以通过其索引引用。例如：

```
>>> m.group(1)
'Malcolm'
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
>>> m.group(2)
'Reynolds'
```

2．match.groups(default=None)

match.groups(default=None)返回(和模式串中的从 1 开始的所有分组)匹配的所有子组的元组(tuple)对象。

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

模式中的第一个分组(\d+)匹配的是 24，而第二个分组(\d+)匹配的是 1632，因此返回的 match 对象的 groups()返回的是对应这些模式分组的子组(subgroups)构成的元组(tuple)对象。

未参与匹配的模式的分组对应返回的 subgroup 子组就是传递给默认参数 default 的值，默认为 None。例如，下面模式中的第二个分组(\d+)在字符串中没有得到匹配，返回的第二个子组的值默认为 None。

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()     #Second group defaults to None.
('24', None)
```


如果将默认值设置为其他值，如“0”，那么返回的子组中第二个子组的值默认就是“0”。例如：

```
>>> m.groups('0')   #Now, the second group defaults to '0'.
('24', '0')
```

match 的 groups()和 group()的区别如下。

- groups()针对的是用()定义的显式分组的模式，返回的是匹配分组的子串构成的元组，它具有与正则表达式模式匹配的所有值。
- group()可针对一般的模式，如果模式中没有用()定义的显式分组，则 group()和 group(0)返回匹配模式的整个子串。如果模式中有分组，那么 group(i)就返回分组 i 匹配的子组。例如：

```
import re
s = 'abc -x p -y q'
m = re.search('-\w+ \w+', s)
print(m.groups())
print(m.group(0))
```

输出：

```
()
-x p
```

因为正则表达式中没有用()定义显式分组，所以其匹配结果为空，而 group()则返回匹配的完整子串。

3．match.start([group])、match.end([group])、match.span([group])

match.start([group])、match.end([group])返回匹配的 group 子组在主串中的开始索引和结束索引。group 默认为零(表示整个匹配的子串)。如果子组存在但没有参与匹配，则返回–1。

match.span([group])返回的是起始位置和结束位置的元组。

对于匹配对象 m 和对该匹配有贡献的组 g，由组 g 匹配的子串在主串中的起始位置和结束位置分别是 m.start(g)、m.end(g)。而 m.span()返回的是起始位置和结束位置构成的元组。例如：

```
str = 'This is the mail-box: xue.pro-8@gmail.com, please mail to us.'
m = re.search(r'([\w.-]+)@([\w.-]+)', str)
print(m.start(1),m.end(1))
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print(m.start(2),m.end(2))
print(m.span(1))
print(m.span(2))
```

输出：

```
22 31
32 41
(22, 31)
(32, 41)
```

10.4.2 标志参数

re 模块的模式匹配函数或 RegexObject 的模式匹配方法通常有一个可选的标志参数(flags)，用于对模式匹配进行不同的控制。这个标志参数默认值是 0，也可以是不同标志通过“或”运算的组合。不同标志及其含义如表 10-5 所示。

表 10-5 标志及其含义

标 志	含 义
re.I 或 re.IGNORECASE	忽略大小写
re.M 或 re.MULTILINE	控制开始/结束的元字符{^, $}对每一行还是整个文本(多行，re.MULTILINE)都起作用
re.S 或 re.DOTALL	使元字符对每一行都起作用
re.U 或 re.UNICODE	根据 Unicode 字符集解释字母，此标志会影响\ w、\ W、\ b、\ B 的行为，即{\w, \W, \b, \B}符合 Unicode 规则
re.L 或 re.LOCALE	根据当前区域设置解释单词，此解释会影响字母组(\ w 和\ W)及字边界行为(\ b 和\ B)
re.X 或 re.VERBOSE	该标志通过灵活的格式以便将正则表达式写得更易于理解。当该标志被指定时，在模式串中的空白符被忽略，除非该空白符在字符集中或在反斜杠之后；它也可以允许将注释写入 re 模块，这些注释会被引擎忽略；注释用#号来标识，但是该符号不能在字符串或反斜杠之后，这样做可以让程序更清晰地组织和缩进

例如：

```
import re
ss = """abc
    def
    ghi"""

r1 = re.findall(r"^\w", ss)        #默认^表示匹配整个文本的开头
r2 = re.findall(r"^\w", ss, flags = re.MULTILINE)  #^表示匹配每一行的开头

print(r1)                          #['a']
print (r2)                         #['a', 'd', 'g']
```

输出：

```
['a']
['a', 'd', 'g']
```

通过 re.X 或 re.VERBOSE 标志，在正则表达式中添加注释说明。例如，下面两行代码的功能是完全一样的：

```
a = re.compile(r"""\d +  #the integral part
                   \.    #the decimal point
                   \d *  #some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

10.5 习题

1. 变量名由字母数字和下画线构成，且第一个字符不能是数字，请编写一个正则表达式判断变量名的这个规则，并测试该正则表达式。
2. 假如邮箱的用户名和服务器名可以包含下画线字符_，且第一个字符不能是数字字符，请改写验证邮箱合法的正则表达式 r'([\w.-]+)@([\w.-]+)以满足这个要求。
3. 定义一个正则表达式用于匹配一个“区号+8 位电话号码”的字符串如“02585810123”，并输入不同的字符串来测试其是否能够正确匹配。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第11章 并发计算

现代计算机的操作系统都是多任务的操作系统，即可以同时运行几个任务。例如，操作系统可以打开多个应用程序，如浏览器、文本编辑器、媒体播放器等，这些应用程序可以同时运行，使得用户可以一边编程一边听音乐。简单地讲，就是操作系统是可以同时运行多个任务的多任务系统。

每当用户执行某个程序时，操作系统就将这个程序加载到内存中运行，并用一个称为“**进程**”(processes)的数据结构管理该程序。因此，“进程”可以看作一个在计算机中执行的程序，一个进程就是一个具体的程序实例。例如，打开一个记事本程序，该程序的一个实例就会加载到计算机中，即产生一个进程，如果再次打开记事本程序，就又产生另一个新的进程，即同一个应用程序可能有多个进程(程序的实例)在计算机中运行。

计算机一般都有多个 CPU，即“多核”，但是因为进程数目往往远远大于 CPU 数目，所以操作系统会按照一定的调度算法，让这些进程轮流获得在 CPU 上运行的机会，称为“**进程管理**”。

进程是一个程序的实例，每个程序实例的代码和数据都占据独立的内存，如果一个程序有多个运行的进程，那么消耗的空间就会很大。但是，进程之间进行通信是很困难的。为了使一个程序能够并发执行，既提高程序运行速度又避免进程占用很大内存空间，人们提出了“**线程**(threads)”概念，即一个进程可以有多个同时执行的“线程”，这些线程可以共享程序代码和数据，但又能执行各自的任务，这样既解决了内存的消耗问题，又可以通过共享的内存在线程之间进行通信。例如，一个浏览器进程中，一个线程下载数据，另一个线程负责刷新页面、与用户交互，从而提高了程序性能。想一想，每次加载图片时，如果浏览器对用户输入停止反应会怎样？

Python 提供了支持进程和线程的模块。例如，multiprocessing、subprocess、signal 等模块提供对多进程的支持，而 threading、_thread 模块提供了对多线程的支持。

11.1 多线程

Python 的 threading 模块是一个高层的多线程模块，它依赖底层的_thread 模块，可通过 threading 模块提供的接口编写多线程程序。

11.1.1 Thread 类

threading 模块的 Thread 类用于表示一个线程。

threading 模块提供下列函数。

- threading.currentThread()：返回当前的线程变量。
- threading.enumerate()：返回一个包含正在运行的线程的 list。“正在运行”是指在线程启动后、结束前的线程，不包括启动前和终止后的线程。
- threading.activeCount()：返回正在运行的线程数量，与 len(threading.enumerate())有相同的结果。

Thread 类提供了以下几种方法。

- run()：用于表示线程活动的方法。
- start()：启动线程活动的方法。
- join([time])：等待至线程终止的方法。对一个线程对象 x，调用 x.join()方法会阻塞调用线程，直至线程 x 因正常退出、抛出未处理的异常终止或超过等待时间为止。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- isAlive()：返回线程是否活动。
- getName()：返回线程名。
- setName()：设置线程名。

1．创建和启动线程

创建一个线程有以下两种方式。

- 直接创建一个 Thread 类对象，并给它的构造函数传递一个**可调用的对象(callable object)**。
- 从 Thread 类定义一个派生类，并重载其构造函数__init__()和__run__()方法，然后创建一个该派生线程类的对象并传递一个可调用的对象。

Thread 类的构造函数规范是：

```
threading.Thread(group=None, target=None, name=None, args=(), kwargs={},
*, daemon=None)
```

其中，参数 target 就是线程的__run__()方法调用的可调用对象，即线程的执行代码，如果不提供，则线程将什么也不做。参数 name 是这个线程的名字，如果不提供，则系统会提供一个默认的名字。参数 args 和 kwargs 分别是 tuple 和 dict 类型的(可变)参数，即传递给 target 对象的参数。

通过 Thread 类对象的 start()方法可以启动一个线程。Thread 的 start()方法会调用 Thread 的__run__()方法执行构造函数的参数 target 指向的可调用对象，同时将参数 args 和 kwargs 传递给这个 target 对象。

下面的代码在主线程中创建了 5 个线程。每个线程的构造函数中接收了一个可调用对象，即函数 worker(args)，在该构造函数中，还传递了一个 tuple 对象(i,)作为函数 worker()的参数 args：

```
#thread_args.py
import threading
def worker(args):
    print('I am a Worker with params %s'%args )

print('This is the main thread!')
threads = []
for i in range(5):
    t = threading.Thread(target=worker,args=(i,))
    threads.append(t)
    t.start()
```

输出：

```
This is the main thread!
I am a Worker with params 0
I am a Worker with params 1
I am a Worker with params 2
I am a Worker with params 3
I am a Worker with params 4
```

2．为线程命名

Thread 的构造函数有一个参数 name，它是用来为线程命名的，如果没有提供这个参数，则系统在创建线程对象时会自动生成一个名字。可以通过这个参数为线程命名，以区分和跟踪线程。例如：

```
#thread_name.py
import threading
import time

def thread_A():
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(2)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    print(threading.current_thread().getName(), 'Exiting')

def thread_B():
    print(threading.current_thread().getName(), 'Starting')
    time.sleep(3)
    print(threading.current_thread().getName(), 'Exiting')

t = threading.Thread(name='thread_A', target=thread_A)
w = threading.Thread(name='thread_B', target=thread_B)
w2 = threading.Thread(target=thread_A)          #使用默认的名字

w.start()
w2.start()
t.start()
```

输出：

```
thread_B Starting
Thread-6 Starting
thread_A Starting
Thread-6 Exiting
thread_A Exiting
thread_B Exiting
```


3．守护进程与非守护进程线程

一个线程可以被设置为是否 **daemen（守护）**线程，Thread 的构造函数的 daemon 参数默认是 None，默认情况下创建的是非守护线程。

一个程序只有一个主线程，即 Python 程序的初始线程，主线程必须等待这些非守护线程都执行完后才能退出。而守护线程不能阻塞主线程，即主线程不用等待守护线程执行完就可以退出。守护线程主要用于不需要用户交互的后台服务或那些即使突然死掉也不造成数据破坏的线程。

可以在创建线程时设置参数“daemon=True”或创建一个线程后调用 set_daemon（True）方法，将该线程设置为守护线程。例如：

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-9s) %(message)s',)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

def daemon():
    logging.debug('Starting')
    time.sleep(10)
    logging.debug('Exiting')

logging.debug('Starting')
t = threading.Thread(name='Non-daemon Thread', target=non_daemon)

d = threading.Thread(name='Daemon Thread', target=daemon,daemon=True)
#或者 d.setDaemon(True)

d.start()
t.start()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    logging.debug('Exiting')
```

输出：

```
(MainThread) Starting
(Daemon Thread) Starting
(Non-daemon Thread) Starting
(MainThread) Exiting
(Non-daemon Thread) Exiting
```

运行该程序，当主线程和所有非守护线程退出后，守护线程并没有立即退出，而是等待 10 秒完成自己工作后才退出，即 10 秒后才输出下列信息：

```
(Daemon Thread) Exiting
```

4．join()方法

如果希望主线程等待守护线程结束，则可在主线程中通过守护线程对象调用 join()方法等待守护线程结束。例如：

```
import threading,time
def non_daemon():
    print('非守护线程-开始')
    print('非守护线程-退出')

def daemon():
    print('守护线程-开始')
    time.sleep(10)
    print('守护线程-退出')

t = threading.Thread(name='非守护线程', target=non_daemon)
d = threading.Thread(name='守护线程', target=daemon,daemon=True)
#或者 d.setDaemon(True)

d.start()
t.start()
d.join()
t.join()
print('守护线程活动状态：', d.isAlive())
print('退出主线程')
```


输出：

```
守护线程-开始
非守护线程-开始
非守护线程-退出
守护线程-退出
守护线程活动状态：False
退出主线程
```

在默认情况下，通过守护线程对象 x 调用 join()方法会无限期地阻塞调用线程等待线程 x 结束。也可以给 join()方法设置一个等待时间，当超过这个时间后，该 join()方法的调用线程就不再等待线程 x 结束。可以在调用线程里，用 alive()方法查询线程 x 是处于活动状态还是停止状态。假设修改上述的“d.join()”为“d.join(1.0)”，则程序运行结果为：

```
守护线程-开始
非守护线程-开始
非守护线程-退出
守护线程活动状态：True
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
退出主线程
守护线程-退出
```

即主线程在守护进程结束前就首先结束了。

5. 枚举所有线程

在 11.1.1 节第一个例子中，用了一个 list 对象的 threads 保存所有线程对象的引用，实际上不必这样做，可以通过 threading 模块的 enumerate() 方法枚举所有活动的线程(包括主线程)。例如：

```
import random,threading,time
def worker():
    r = random.randint(1,6)/10
    print('%s 睡眠 %0.2f'%(threading.current_thread().getName(),r))
    time.sleep(r)
    print('退出子线程%s'%threading.current_thread().getName())

for i in range(3):
    t = threading.Thread(target=worker, name=str(i),daemon=True)
    t.start()

main_thread = threading.main_thread()      #获得主线程的句柄(引用)
for t in threading.enumerate():
    if t is main_thread:
        continue
    print('joining 的子线程 %s'%(t.getName()))
    t.join()
```

输出：

```
0 睡眠 0.40
1 睡眠 0.50
2 睡眠 0.50joining 的子线程 SockThread

退出子线程 0
退出子线程 1
退出子线程 2
```

6. 定义 Thread 的派生类

也可以从 Thread 线程类定义一个派生类，然后创建这个派生类的对象。派生类需要重载__init__() 方法和__run__() 方法。例如：

```
import threading
class DerivedThread(threading.Thread):
    def __init__(self,args):
        super(DerivedThread, self).__init__()
        self.__args__ = args

    def run(self):
        print('子线程的参数：%s'%self.__args__ )
DerivedThread((1,)).start()
DerivedThread((2,)).start()
DerivedThread((3,)).start()
print('退出主线程!')
```

输出：

```
子线程的参数：1
子线程的参数：2
子线程的参数：3 退出主线程!
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

DerivedThread类的构造函数中只有一个参数args。通常，派生类的构造函数的参数可以和Thread的构造函数的参数一样。例如：

```
#DerivedThread2.py
import threading
class DerivedThread(threading.Thread):
    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, *, daemon=None):
        super().__init__(group=None, target=target, name=name,
                         daemon=daemon)
        self.args = args
        self.kwargs = kwargs

    def run(self):
        print('子线程的参数：%s and %s'%(self.args, self.kwargs))
DerivedThread((1,), kwargs={'name': '王', 'age': 21 }).start()
DerivedThread((2,), kwargs={'name': '张', 'age': 22 }).start()
DerivedThread((3,), kwargs={'name': '李', 'age': 23 }).start()
print('退出主线程!')
```

输出：

```
子线程的参数：() and {'name': '王', 'age': 21}
子线程的参数：() and {'name': '张', 'age': 22}
子线程的参数：() and {'name': '李', 'age': 23}退出主线程!
```


7．Timer 线程

Timer 线程是 Thread 类的一个派生类，它通过给构造函数传递一个延迟参数，可以创建一个延迟一定时间再启动的线程，在延迟期间内，可以通过调用函数 cancel()取消这个线程的执行。例如：

```
#thread_timer.py
import threading,time,logging

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

def f():
    logging.debug('f running')

def g():
    logging.debug('g running')

t1 = threading.Timer(3.0, f)       #创建 Timer 线程，延迟 3.0 秒
t1.setName('t1')
t2 = threading.Timer(5.0, g)       #创建 Timer 线程，延迟 5.0 秒
t2.setName('t2')

t1.start()
t2.start()

time.sleep(0.2)
logging.debug('取消 %s', t2.getName())
t2.cancel()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
(MainThread) 取消 t2
(t1        ) f running
```

11.1.2 线程同步

因为多个线程可以修改同一数据，所以有可能破坏数据的正确性，为此需要对访问同一数据的多个线程进行同步。

Python 提供了不同的“同步原子”来保证线程的同步，主要包括 Lock、RLock、Semaphore、Event、Condition 和 Barrier，当然用户可以从这些类派生出子类以定制自己的同步原子。

1．Lock(锁)和 RLock(重入锁)

可以通过调用 Lock 对象或 RLock 对象的 acquire()方法和 release()方法来获得或释放一个“互斥锁”对象，以保证每次只有一个线程能够访问共享对象。例如：

```
import threading, time, logging

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

count = 0
lock = threading.Lock()            #创建一个互斥锁对象

class CountThread(threading.Thread):
    def run(self):
        global count
        time.sleep(1)

        if lock.acquire(1):        #lock.acquire()获得互斥锁
            count += 1
            logging.debug('%s: count 值: %s',self.name,count)
            lock.release()         #lock.release()释放互斥锁

for i in range(5):
   t = CountThread(name = 'Thread '+str(i))
   t.start()
```

输出：

```
(Thread 0  ) Thread 0: count 值: 1
(Thread 1  ) Thread 1: count 值: 2
(Thread 2  ) Thread 2: count 值: 3
(Thread 3  ) Thread 3: count 值: 4
(Thread 4  ) Thread 4: count 值: 5
```

上面程序使用的是全局变量作为多个线程的共享数据 count，也可以将共享数据作为参数传递给线程对象的构造函数。例如：

```
import threading, time, logging, random

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)
#Counter 里保存了共享数据 self.value 和互斥锁 self.lock
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
class Counter:
    def __init__(self, start=0):
        self.lock = threading.Lock()
        self.value = start

    def increment(self):
        logging.debug('等待锁 lock')
        self.lock.acquire()
        try:
            logging.debug('获得了锁 lock')
            self.value = self.value + 1
            #logging.debug('count value is %s',self.value)
        finally:
            self.lock.release()

counter = Counter()

def thread_pro(c):                              #接收一个共享参数 c
    for i in range(2):
        pause = random.random()
        logging.debug('睡眠 %0.02f', pause)
        time.sleep(pause)
        c.increment()
        logging.debug('count 值: %s',c.value)
    logging.debug('线程结束')

for i in range(3):
   t = threading.Thread(target = thread_pro, name = 'Thread '+str(i),args=(counter,))
   t.start()
```


输出：

```
(Thread 0  ) 睡眠 0.26
(Thread 1  ) 睡眠 0.82
(Thread 2  ) 睡眠 0.45
(Thread 0  ) 等待锁 lock
(Thread 0  ) 获得了锁 lock
(Thread 0  ) count 值: 1
(Thread 0  ) 睡眠 0.37
(Thread 2  ) 等待锁 lock
(Thread 2  ) 获得了锁 lock
(Thread 2  ) count 值: 2
(Thread 2  ) 睡眠 0.23
(Thread 0  ) 等待锁 lock
(Thread 0  ) 获得了锁 lock
(Thread 0  ) count 值: 3
(Thread 0  ) 线程结束
(Thread 1  ) 等待锁 lock
(Thread 1  ) 获得了锁 lock
(Thread 1  ) count 值: 4
(Thread 1  ) 睡眠 0.78
(Thread 2  ) 等待锁 lock
(Thread 2  ) 获得了锁 lock
(Thread 2  ) count 值: 5
(Thread 2  ) 线程结束
(Thread 1  ) 等待锁 lock
(Thread 1  ) 获得了锁 lock
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
(Thread 1  ) count 值：6
(Thread 1  ) 线程结束
```

一个线程只能请求一次 Lock 对象，如果一个线程有多处代码要请求一个 Lock 对象，那么除第一次外，其他的请求都会失败，即使传递的等待时间为 0。例如：

```
import threading
lock = threading.Lock()
print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))
```

输出：

```
First try : True
Second try: False
```

为了能够多次请求一个“锁”，可以用 RLock 对象代替 Lock 对象。除可以一个线程多次请求 RLock 对象外，使用 RLock 对象和使用 Lock 对象是没有区别的，唯一需要注意的是，RLock 对象里有一个计数器，用于记录请求次数，相应地，也要调用同样次数的 release()才能完全释放 RLock，使其他线程获得 RLock 的使用权。例如：

```
import threading
lock = threading.RLock()
print('First try :', lock.acquire())
print('Second try:', lock.acquire(0))
```


输出：

```
First try : True
Second try: True
```

Lock 和 with 语句是相容的，所以可以用 with 语句简化 Lock 对象的使用，以保证 Lock 对象自动获取和安全释放。例如：

```
#thread_lock_with.py
import threading
import logging

logging.basicConfig(
    level=logging.DEBUG,
    format='(%(threadName)-10s) %(message)s',
)

def f_with(lock):
    with lock:
        logging.debug('使用 with 获取 Lock 对象')


def f_no_with(lock):
    lock.acquire()
    try:
        logging.debug('直接获取 Lock 对象')
    finally:
        lock.release()          #必须使用 release()释放这个互斥锁

lock = threading.Lock()
wt = threading.Thread(target=f_with, args=(lock,))
nwt = threading.Thread(target=f_no_with, args=(lock,))

wt.start()
nwt.start()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
(Thread-6  ) 使用 with 获取 Lock 对象
(Thread-7  ) 直接获取 Lock 对象
```

2．事件(Event)

Event 是一个简单的线程间通信的同步原子。一个 Event 对象内部维护一个标志，可以通过函数 set()、函数 clear()设置或清除一个线程，其他线程可以通过 wait()方法等待这个 Event 被设置为 True，即调用 wait()方法的线程会一直阻塞，直到 Event 对象标志为 True。

例如：

```
import random, time, threading
event = threading.Event()

def waiter(event, n):
    for i in range(n):
      print("%s. 开始等待标志为 True..." % (i+1))
      event.wait()              #阻塞直到标志为 True.
      print("等待完成时间为:", time.ctime())
      event.clear()             #清除标志

def setter(event, n):
    for i in range(n):
      time.sleep(random.randrange(2, 5))
      event.set()               #设置标志

n = random.randrange(2,5)
w = threading.Thread(target=waiter, args=(event, n))
s = threading.Thread(target=setter, args=(event, n))
w.start()
s.start()
w.join()
s.join()
print("主线程结束.")
```

输出：

```
1. 开始等待标志为 True...
等待完成时间为: Thu Oct  4 16:56:53 2018
2. 开始等待标志为 True...
等待完成时间为: Thu Oct  4 16:56:55 2018
3. 开始等待标志为 True...
等待完成时间为: Thu Oct  4 16:56:57 2018
4. 开始等待标志为 True...
等待完成时间为: Thu Oct  4 16:57:01 2018
主线程结束.
```

3．条件(Condition)

Condition 对象是 Event 对象的高级版本，也用于进程间通信，并且可以用函数 notify()通知其他线程的某个状态发生了变化。例如，通知某个资源已经具备(如网络已经连接、数据下载已经完成

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

等)。其他线程在等待这个 Condition 对象前必须首先获得 acquire()这个 Condition 对象(因此它也是一个锁)，一个线程也应该通过 release()释放这个 Condition 对象，以便其他线程能够通过 acquire()获得 Condition 对象。

例如，著名的“生产者-消费者”问题，生产者不断生产一些产品，而消费者不断消费这些产品：

```
import random, time,threading

condition = threading.Condition()
products = []

def producer(products, nitems):
    for i in range(nitems):
        time.sleep(random.randrange(2, 5))  #Sleeps for some time.
        condition.acquire()
        num = random.randint(1, 100)
        products.append(num)          #将产品放入产品列表供消费者消费
        condition.notify()            #通知消费者有产品
        print("%s: 生产了一个新产品: %s"%(time.ctime(), num))
        condition.release()

def consumer(products, nitems):
    for i in range(nitems):
        time.sleep(random.randrange(2, 5))
        condition.acquire()
        condition.wait()              #阻塞，直到有一个产品可以消费
        print("%s: 消费一个产品: %s"%(time.ctime(), products.pop()))
        condition.release()

threads = []
nloops = random.randrange(2, 4)
nloops2 = random.randrange(2, 4)
threads.append(threading.Thread(target=producer, args=(products, nloops,)))
threads.append(threading.Thread(target=consumer, args=(products, nloops2,)))
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
print("退出主线程")
```

输出：

```
Thu Oct  4 13:27:15 2018: 生产了一个新产品: 22
Thu Oct  4 13:27:19 2018: 生产了一个新产品: 85
Thu Oct  4 13:27:19 2018: 消费一个产品: 85
Thu Oct  4 13:27:22 2018: 生产了一个新产品: 50
Thu Oct  4 13:27:22 2018: 消费一个产品: 50
退出主线程
```

4．障碍(Barrier)

Barrier 也是一种简单的线程间同步原子，每个调用 Barrier 对象的 wait()方法的线程都会阻塞，只有等待所有调用该对象的 wait()方法的线程都结束后，这个 Barrier 条件才满足，这些线程就可以一起开始后续的执行过程。如同只有当所有等待乘一辆车的乘客都上车后，这辆车才开始出发一样，否则其他乘客都要一起等待。例如：

```
import threading, time, random
num = 2
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
barrier = threading.Barrier(num+1)

def worker():
    time.sleep(random.randrange(2, 10))
    name = threading.current_thread().getName()
    print("%s reached the Barriers at: %s" % (name, time.ctime()))
    barrier.wait()

for i in range(num):
    t = threading.Thread(name=str(i), target=worker)
    t.start()

barrier.wait()    #主线程也等待 Barriers

print("Exit!")
```

输出：

```
1 reached the Barriers at: Thu Oct  4 12:45:19 2018
0 reached the Barriers at: Thu Oct  4 12:45:24 2018
Exit!
```

5．信号灯(Semaphore)

Semaphore 是计算机科学历史上最古老的同步原子之一，它由荷兰计算机科学家 Edsger W. Dijkstra 发明。Semaphore 管理一个内部计数器，acquire()会使计数器减少，而 release()会使计数器增加。计数器永远不会小于 0，当一个调用 acquire()的线程计数器为 0 时，该线程就会阻塞，并等待其他线程调用 release()释放该线程。下列代码用 Semaphore 解决“生产者-消费者”问题：

```
import random, time,threading

container = threading.BoundedSemaphore(3) #仓库中最多能够存放三件商品
def producer(nloops):
     for i in range(nloops):
       time.sleep(random.randrange(2, 5))
       print(time.ctime(), end=": ")
       try:
          container.release()
          print("生产了一件产品")
       except ValueError:
          print("库存已满！")
def consumer(nloops):
    for i in range(nloops):
       time.sleep(random.randrange(2, 5))
       print(time.ctime(), end=": ")
       if container.acquire(False):
          print("消费一件产品")
       else:
          print("库存已空！")
threads = []
nloops = random.randrange(2, 4)
nloops2 = random.randrange(2, 4)
threads.append(threading.Thread(target=producer, args=(nloops,)))
threads.append(threading.Thread(target=consumer, args=(nloops2,)))
for thread in threads:
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        thread.start()
    for thread in threads:
        thread.join()
    print("退出主线程")
```

输出：

```
Thu Oct  4 13:07:32 2018: 库存已满!
Thu Oct  4 13:07:34 2018: 消费一件产品
Thu Oct  4 13:07:36 2018: 消费一件产品
Thu Oct  4 13:07:36 2018: 生产了一件产品
Thu Oct  4 13:07:39 2018: 消费一件产品
退出主线程
```

11.2 多进程

11.2.1 创建进程

Multiprocessing 模块的 Process 类表示进程，可通过创建 Process 类对象来创建一个进程，并通过关键字参数 args 将一个 tuple 中的参数传递给进程函数 worker()。例如：

```
import multiprocessing

def worker(arg):
    print("Hello, I am subprocess " ,arg)

if __name__ == "__main__":
    print('I am parent process')
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        jobs.append(p)
        p.start()
```

输出：

```
I am parent process
```

在操作系统的命令行运行上述程序，将输出下面的结果：

```
I am parent process
Hello, I am subprocess 1
Hello, I am subprocess 0
Hello, I am subprocess 4
Hello, I am subprocess 2
Hello, I am subprocess 3
```

可以看出，这些进程的执行顺序是不确定的，这是竞争使用 CPU 的结果。

在上述代码中，子进程的函数代码和主进程的代码在同一脚本文件中，通常将子进程的代码放在另外一个单独的脚本文件中。例如：

```
#multiprocessing.py
import multiprocessing
import worker

if __name__ == '__main__':
    jobs = []
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        for i in range(5):
            p = multiprocessing.Process(target=worker.worker,args=(i,))
            jobs.append(p)
            p.start()
    #worker.py
    def worker(arg):
        print('Hello, I am subprocess ',arg)
        return
```

11.2.2 从 Process 类派生自己的进程类

用户也可以首先从 Process 类派生自己的进程类，然后创建自己的进程类对象。如果要从 Process 类派生自己的进程类，则需要重载构造函数__init__()方法和 run()方法，run()方法里就是该进程的工作代码。例如：

```
    import multiprocessing
    class MyProcess(multiprocessing.Process):
        def __init__(self, arg):
            super(MyProcess, self).__init__()
            self.__arg__ = arg

        def run(self):
            print('Hello, I am subprocess ' + str(self.__arg__))

    if __name__ == "__main__":
        print('I am parent process')
        jobs = []
        for i in range(5):
            p =MyProcess(i)
            jobs.append(p)
            p.start()
```


在命令行执行该程序的运行输出结果：

```
    I am parent process
    Hello, I am subprocess 3
    Hello, I am subprocess 1
    Hello, I am subprocess 0
    Hello, I am subprocess 2
    Hello, I am subprocess 4
```

11.2.3 为进程命名

每个进程被创建时，都会自动为进程分配一个 id 以便区分不同的进程，因此，不需要为进程命名。但是，如果想跟踪该进程，则为不同进程命名也是可以的。例如：

```
    import multiprocessing
    import time

    def worker():
        name = multiprocessing.current_process().name
        print(name, 'Starting')
        time.sleep(2)                              #睡眠一小段时间
        print(name, 'Exiting')

    def my_service():
        name = multiprocessing.current_process().name
        print(name, 'Starting')
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        time.sleep(3)
        print(name, 'Exiting')


    if __name__ == '__main__':
        service = multiprocessing.Process(
            name='my_service',
            target=my_service,
        )
        worker_1 = multiprocessing.Process(
            name='worker 1',
            target=worker,
        )
        worker_2 = multiprocessing.Process(    #默认名字
            target=worker,
        )

        worker_1.start()
        worker_2.start()
        service.start()
```

输出结果：

```
worker 1 Starting
worker 1 Exiting
Process-3 Starting
Process-3 Exiting
my_service Starting
my_service Exiting
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 12 章　图形用户接口(GUI)编程

Python 提供了多个用于开发图形用户界面 GUI 程序的库，如 Tkinter、wxPython、PyQt、Kivy、PyGTK、PyKDE4、PyOpenGL 等。

本章主要介绍用 Python 自带的标准 GUI 库 Tkinter 开发 GUI 应用程序，Tkinter 提供了强大的面向对象的接口。

12.1　Tkinter 基础

12.1.1　事件驱动编程

Tkinter 是 Tk GUI toolkit 的 Python 标准接口，是一个面向对象的事件驱动的 GUI 库，其用户界面是由一些不同类型的图形元素(也称**部件**或**控件**)组成的。当用户和用户界面上的部件交互时，就会触发不同的事件(如单击按钮、键盘输入、网络连接已建立或丢失)。当这些事件发生时，应用程序应该做出适当的反应(响应)，通常是调用相应的函数来执行相关的动作，这个函数也习惯地被称为“**事件处理器函数**”或“**事件处理器**”。

用 Tkinter 库创建一个 GUI 程序的主要步骤如下。

(1)导入 Tkinter 模块。

(2)创建 GUI 应用程序主窗口。

(3)将一个或多个图形界面元素(也称“部件”或“控件”)添加到 GUI 应用程序中。

(4)进入主事件循环：检测触发的事件，并调用事件处理函数(事件处理器)执行相应的动作。

下面是基于 Tkinter 的 GUI 程序的结构：

```
import tkinter                  #导入 Tkinter 模块
window = tkinter.Tk()           #创建 GUI 应用程序主窗口
#添加部件 widgets，设置事件处理器...
window.mainloop()               #进入主事件循环
```

12.1.2　第一个 GUI 程序

第一个 GUI 程序(如图 12-1 所示)是带有一个标签(Label)和两个按钮(Button)的窗口：

```
import tkinter                  #导入 Tkinter 模块
window = tkinter.Tk()           #创建 GUI 应用程序主窗口

#添加部件 widgets，设置事件处理器...
lbl = tkinter.Label(window, text="First GUI App!")
ok_button = tkinter.Button(window, text="OK")
cancel_button = tkinter.Button(window, text="Cancel", command = window.quit)
lbl.pack()
ok_button.pack()
cancel_button.pack()

window.mainloop()               #进入主事件循环
```

tkinter 的 Tk()方法 tkinter.Tk()构造了一个代表应用程序主窗口的根窗口对象，一个应用程序只

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

有一个根窗口，除可创建根窗口外，还可以创建其他窗口。tkinter.Tk()方法构造了一个代表引用程序窗口的根窗口对象。

Button 和 Label 分别是表示按钮和标签的类，这些部件(widget)都必须有一个父部件，程序中将主窗口对象 window 作为它们的父部件，所以它们都是父部件的孩子。因此，在创建标签 Label 类的对象 lbl 和按钮 Button 类的对象 ok_button 和 cancel_button 时，传递给构造函数的第一个参数就是父空间对象的引用，即 window。

在每个部件上使用布局方法 pack()可将其放置在其父部件中。

在构造按钮 cancel_button 时使用了关键字参数 command 来指定处理该按钮的鼠标点击事件的函数，这里将 window 部件的 quit()方法作为该按钮的事件处理函数，该函数的作用就是关闭窗口并退出整个程序。而 ok_button 按钮没有给它设置点击事件处理函数。因此，当点击该按钮时，将不会有任何反应。三个部件通过 text 关键字参数设置文本。

图 12-1　第一个 GUI 程序

window.mainloop()是应用程序的主窗口的方法，此方法将永久循环：等待事件并调用事件处理函数处理事件，直到用户退出程序(如关闭窗口，或者通过控制台中的键盘中断终止程序)。

12.1.3　Tkinter 部件

Tkinter 提供了许多从 Widget 类派生出来的各种部件类，现举例如下。

- 框架(Frame)：容纳其他部件的容器部件，它用于在应用程序的布局中将相关的部件分组在一起。
- 顶层窗口(Toplevel)：这是一个显示为单独窗口的容器部件。
- 画布(Canvas)：这是绘制图形的部件。在高级用法中，它也可以用来创建自定义部件，因为可以在其中绘制任何东西，并使其可以交互。
- 文本(Text)：显示格式化的文本，可以编辑并可以嵌入图像。
- 标签(Label)：这是一个简单的部件，它显示一小段文字或图像，但通常不是交互式的。
- 消息(Message)：与标签类似，但是用于包装更长的文本。
- 按钮(Button)：通常直接映射到用户操作上，当用户点击按钮时，会触发一些事件。
- 输入框(Entry)：用户可以输入单行文本。
- 列表框(Listbox)：向用户提供一列选项。
- 复选框，又称多选按钮(Checkbutton)：显示多个选项，用户可以一次选择多个选项。
- 单选框(Radiobutton)：显示多个选项，但用户一次只能选择其中一个选项。
- 滑动条(Scale)：让用户拖动滑动条选择一个最大值和最小值之间的数值。
- 滚动条(Scrollbar)：允许用户在浏览时滚动因太大而无法一次显示的内容。
- 菜单(Menu)和菜单按钮(Menubutton)：用于创建菜单和菜单按钮。

12.1.4　布局——几何管理

第一个 GUI 程序中的 GUI 具有相对简单的布局，即使用 pack()方法将三个部件排列在父窗口的单个列中。pack()方法是 Tkinter 中可用的三种不同几何管理器(geometry managers)之一，另外两个是 grid()和 place()。

1．pack()方法

pack()方法的作用是将部件放置在父窗口里，并将它们组织在一个块中。pack()方法默认将部件从上到下垂直地排列在一起，其格式是：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    widget.pack(pack_options)
```

它有三个选项(pack_options)。

- side：控制部件在父部件的相对位置，包括 TOP(顶部)、BOTTOM(底部)、LEFT(左边)、RIGHT(右边)，默认值是 TOP。
- fill：确定部件是否填充由布局管理器分配给它的任何额外空间。默认值是 NONE(保持它自己的最小尺寸)，其值也可以是 X(仅填充水平)、Y(仅填充垂直)、BOTH(填充水平和垂直)。
- expand：当设置为 True 时，窗口部件将展开填充窗口部件的父窗口中未使用的所有空间。

假如修改上面程序中 pack()的参数如下：

```
lbl.pack(fill=tkinter.X)
ok_button.pack(fill=tkinter.X)
cancel_button.pack(fill=tkinter.X)
```

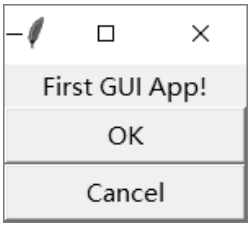
将产生如图 12-2(a)所示的界面。

下列代码产生 12-2(b)所示的界面：

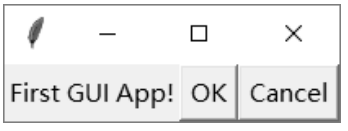
```
lbl.pack(side=tkinter.LEFT)
ok_button.pack(side=tkinter.LEFT)
cancel_button.pack(side=tkinter.LEFT)
```


下列代码产生 12-2(c)的布局：

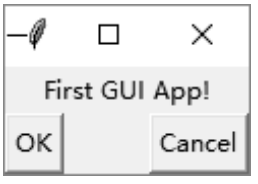
```
lbl.pack()
ok_button.pack(side=tkinter.LEFT)
cancel_button.pack(side=tkinter.RIGHT)
```



(a) fill：X(水平填充)



(b) side：LEFT(靠左)



(c) side：LEFT(靠左)；RIGHT(靠右)

图 12-2　pack()布局

2．grid()方法

grid()网格管理器在父窗口部件中以类似表格的结构组织窗口部件。grid()网格管理器的使用方法很简单，创建控件后，只要调用 grid()方法，告诉布局管理器该控件在表格中的位置(第几行或第几列)即可。如果没有指定行或列，则按调用的顺序自动给控件一个相应的行或列。无须指定网格单元格的大小，布局管理器会自动根据窗口中的控件计算每个网格单元格的大小。

部件被放置在网格的每个单元格中，如果部件本身比单元格小，则可以通过 sticky 指定其在单元格的东(E)、南(S)、西(W)、北(N)的位置。

上面的一个标签和两个按钮可以用如下代码，将每个控件放置在网格的单元格中：

```
lbl.grid(columnspan=2, sticky=tkinter.W)        #跨越两列，靠西
ok_button.grid(row=1)                           #位于 row=1 第二行
cancel_button.grid(row=1, column=1)             #位于 row=1 第二行 column=1 第二列
```

网格的行(row)或列(column)的值默认都是从“0”开始。第一个标签 lbl 没有指定行和列，其所在单元格的行或列默认都为“0”。Ok_button 的行为 1，没有指定列，其所在列默认值就是“0”，

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

cancel_button 的行或列都指定为“1”，因此它和 ok_button 都在“row=1”这一行，但排在其右边，其界面如图 12-3 所示。

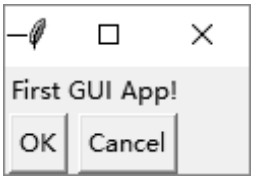


图 12-3　grid()布局

3．place()方法

place()方法允许为部件提供明确的大小和位置。为每个元素指定一个绝对位置是非常不灵活且非常耗时的，因此不推荐使用这种分法。

12.1.5　属性

每个部件都有一些属性(attributes)，下面是所有部件共有的一些标准属性。

- 尺寸(Dimensions)：如宽(width)、高(height)。
- 颜色(Colors)：如背景颜色(bg)、前景颜色(fg)。
- 字体(Fonts)：其值包括 TkTooltipFont、TkDefaultFont、TkTextFont。
- 填充(Padding)：如 padx、pady 属性表示水平或垂直方向的填充方式。
- 锚点(Anchors)：anchor 属性用于定义文本相对于参考点的位置，其可能值如图 12-4 所示。
- 浮雕样式(Relief styles)：relief 属性指定边界装饰方式，其可能值包括 SUNKEN、RAISED、GROOVE、RIDGE、FLAT，如图 12-5 所示。
- 位图(Bitmaps)：bitmap 属性用于显示一个位图。
- 游标(Cursors)：cusor 属性用于显示鼠标指针所在的位置的小图标。
- 状态(State)：如状态属性 state，其值包括 NORMAL、ACTIVE、DISABLED。

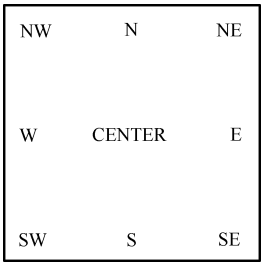


图 12-4　anchor 属性的可能值

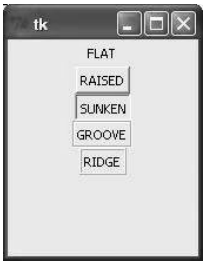


图 12-5　relief 属性的可能值

这些属性既可以在创建部件时设置，也可以用方法设置。例如，下列代码的 Label 和 Button 部件在创建时指定了一些属性(如图 12-6 所示)：

```
window.geometry('200x120')
lbl = tkinter.Label(window, text="标签",bg="orange", fg="red")  #文本、背景和前景颜色
cancel_button = tkinter.Button(window, text="Cancel", command = window.quit)
```

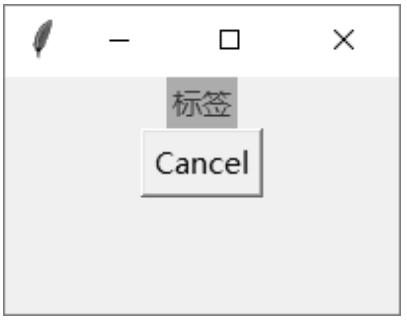


图 12-6　创建部件时指定属性

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

12.1.6 自定义事件处理函数

添加控件后，还要给相应控件添加事件处理功能。例如，给按钮 ok_button 添加一个“点击事件”处理功能，首先需要定义一个事件处理函数，然后让该控件的 command 属性引用这个事件处理函数。当用户点击该按钮时，该函数修改标签控件 lbl 的文本内容：

```
#button 的事件处理函数
def ok_clicked():
    lbl.configure(text="Button was clicked !!")

#创建 button 对象时，让其 command 属性引用这个事件处理函数
ok_button = tkinter.Button(window, text="OK", command=ok_clicked)
```

下面是完整的程序：

```
import tkinter                          #导入 Tkinter 模块

window = tkinter.Tk()                   #创建 GUI 应用程序主窗口
window.title("My First GUI App")
window.geometry('200x120')

#添加部件 widgets...
lbl = tkinter.Label(window, text="标签",bg="orange", fg="blue")  #文本、背景和前景颜色
lbl.grid(column=0, row=0)               #设置位置

#button 的事件处理函数
def clicked():
    lbl.configure(text="Button was clicked !!")

#创建 button 对象时，让其 command 属性引用这个事件处理函数
btn = tkinter.Button(window, text="Click Me", command=clicked)
btn.grid(column=1, row=0)

window.mainloop()                       #进入主事件循环
```

运行“按钮点击事件”程序的界面如图 12-7 所示。

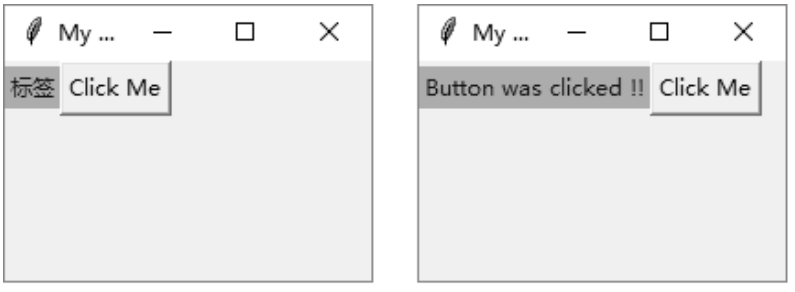


图 12-7 “按钮点击事件”程序的界面

12.1.7 定制事件处理函数

到目前为止，事件处理函数都绑定为默认情况下在 Tkinter 中内置定义的事件，如 Button 类内置了按钮点击，因为点击是普通按钮的预期行为。然而，也可以对任意部件通过绑定方法 bind()，将监听特定事件的自定义事件处理函数绑定到一个控件事件上。

事件由字符串格式的序列名唯一标识。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- <Button-1>、<Button-2>和<Button-3>表示在窗口部件上按下特定鼠标按钮的事件。其中，<Button-1>是鼠标左键，<Button-3>是右键，<Button-2>是中键。需要注意的是，并非所有鼠标都有中键。
- <ButtonRelease-1>、<ButtonRelease-2>、<ButtonRelease-3>分别表示鼠标左键、中键、右键已释放。
- <B1-Motion>、<B2-Motion>、<B3-Motion>分别表示在按下鼠标左键、中键、右键的同时移动了鼠标。
- <Double-Button-1>表示鼠标左键被点击了两次。
- <Enter>和<Leave>表示鼠标光标进入或离开部件的事件。
- <FocusIn>和<FocusOut>分别表示键盘焦点进入或离开部件及其子部件。
- <Return>表示按下了回车键，类似的特殊键还有 Cancel（Break 键）、BackSpace（回退键）、Tab（Tab 键）、Shift_L（Shift 键）、Control_L（Control 键）、Alt_L（Alt 键）、Pause、Caps_Lock、Escape、Prior（Page Up）、Next（Page Down）、End、Home、Left、Up、Right、Down、Print、Insert、Delete、F1、F2、F3、F4、F5、F6、F7、F8、F9、F10、F11、F12、Num_Lock 和 Scroll_Lock 等。
- <Shift-Up>表示用户按下了向上箭头键的同时也按下了 Shift 键。还有和 Down、Left、Right 箭头键结合的组合键。类似的前缀还有 Alt、Shift 和 Control。
- <Key>表示按下键盘上的任意键。事件对象的 char 成员记录了这个按键的值。
- <Configure>表示部件的大小发生了改变的事件，事件对象中包含了表示部件大小属性的 width 和 height。

例如，下面的程序通过“lbl.bind("<Button-1>", cycle_label_text)”给标签 lbl 绑定了鼠标左键按下的事件处理函数 cycle_label_text()。鼠标左键点击标签时，将触发事件处理函数 cycle_label_text()，该函数循环修改 StringVar 类型的变量 label_text 的值，而标签 lbl 的 textvariable 引用的就是这个 tkinter 控制变量 label_text，这使得 label_text 发生修改时，lbl 的 textvariable 也得到了更新。

图 12-8 是当鼠标左键点击标签 lbl 时，标签文本的内容循环变化的截图。

```
import tkinter                  #导入 Tkinter 模块
window = tkinter.Tk()           #创建 GUI 应用程序主窗口

LABEL_TEXT = [
       "第 1 个鼠标点击!",
       "第 2 个鼠标点击!",
       "第 3 个鼠标点击!",
       "继续，将回到循环开头",
   ]
label_index = 0
label_text = tkinter.StringVar(window)
label_text.set(LABEL_TEXT[label_index])

def cycle_label_text(event):
       global label_index
       global label_text
       label_index += 1
       label_index %= len(LABEL_TEXT) #wrap around
       label_text.set(LABEL_TEXT[label_index])


#添加部件 widgets，设置事件处理器...
lbl = tkinter.Label(window, textvariable=label_text)
lbl.bind("<Button-1>", cycle_label_text)   #定制事件处理函数
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
lbl.grid(columnspan=2, sticky=tkinter.W)


#button 的事件处理函数
def ok_clicked():
    lbl.configure(text="Button was clicked !!")

#创建 button 对象时，让其 command 属性引用这个事件处理函数
ok_button = tkinter.Button(window, text="OK", command=ok_clicked)
ok_button.grid(row=1)

cancel_button = tkinter.Button(window, text="Cancel", command = window.quit)
cancel_button.grid(row=1, column=1)


window.mainloop()          #进入主事件循环
```


其中，tkinter.StringVar(window)定义了一个 tkinter 控制变量，当该变量 label_text 的值变化时，将通知其关联部件 window，window 部件将更新其中的标签 lbl 的文本。

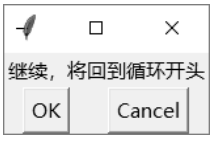


图 12-8　给标签绑定鼠标点击事件

12.1.8　文本输入框

文本输入框(Entry)可以输入文本。例如，如果用户登录时需要输入用户名和密码，则可以编写如下程序：

```
from tkinter import *
window = Tk()

Label(window, text="Username:").grid(row=0)
Label(window, text="Password:").grid(row=1)

username_et = Entry(window)
password_et = Entry(window,show="*", width=15)  #隐藏密码，限制宽度

username_et.grid(row=0, column=1)
password_et.grid(row=1, column=1)

def ok_clicked():
    res = username_et.get() +',你的密码是：'+  password_et.get()
    lbl.configure(text= res)

ok_btn = Button(window, text="登录", command=ok_clicked)
cancel_btn = Button(window, text="取消", command = window.quit)
ok_btn.grid(row=2, column=0)
cancel_btn.grid(row=2, column=1)

lbl = Label(window, text="   ",bg="orange")
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    lbl.grid(row=3)

    window.mainloop()
```

用户登录界面如图 12-9 所示。

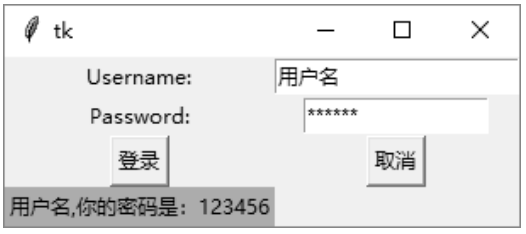


图 12-9　用户登录界面

12.1.9　获取焦点

为避免每次运行程序时都需要点击文本框才能进行文本输入，可以通过 Entry 的 focus() 方法使程序的焦点一开始就处于输入文本框 username_et 上，其形式如下：

```
    username_et.focus()
```


12.1.10　聊天对话框

假如编写一个类似聊天软件那样的简单聊天对话框，如图 12-10 所示，对话框中包含输入文本的 Entry、发送消息的 Button 按钮、显示聊天信息的滚动文本 ScrolledText 或 Label。

聊天信息用 ScrolledText 是最自然的，但即使用最简单的标签 Label 也能显示滚动消息。例如：

```
    from tkinter import *

    window = Tk()
    window.geometry('350x200')

    #justify 文本对齐方式，anchor 锚点部件位置
    messages = Label(window,bg="orange",anchor='w',justify=LEFT)
    messages.pack(side=TOP, expand=1,fill=BOTH)

    input_user = StringVar()
    input_field = Entry(window, text=input_user)
    input_field.pack(side=TOP, fill=X)
    input_field.focus()

    def send_clicked():
        input_get = input_field.get()
        messages['text'] = messages['text']+'\n'+input_get
        input_user.set('')

    send_btn = Button(window, text="发送", command=send_clicked)
    send_btn.pack()

    window.mainloop()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

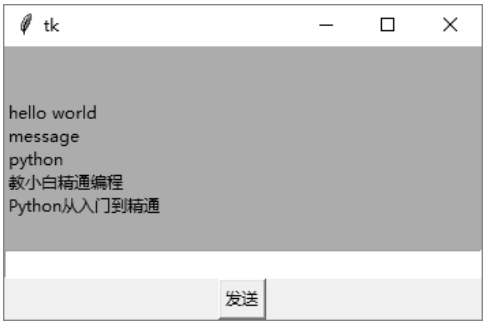


图 12-10　聊天对话框

12.1.11　框架

可以将多个 widget 用一个框架(frame)组合在一起，如将 Entry 输入框和 Button 按钮组合在一个框架中，而将显示消息的 Label 组合在另外一个框架中，再将这两个框架作为整个 window 的子部件。例如：


```
from tkinter import *

window = Tk()
window.geometry('350x200')

#添加一个 frame,用于放置显示消息的标签 Label
frame = Frame(window, relief=RAISED, borderwidth=2)

#创建一个标签 messages，其父控件是 frame
#justify 文本对齐方式，anchor 锚点部件位置
messages = Label(frame,bg="orange",anchor='nw',justify=LEFT)
messages.pack(side=TOP, expand=1,fill=BOTH)


#添加一个 frame2,用于放置输入 Entry 控件和发送按钮 Button
frame2 = Frame(window)
frame2.pack(side=BOTTOM,fill=X, expand=False)
frame.pack(fill=BOTH, expand=TRUE)

#创建输入信息的 Entry 控件 input_field
input_user = StringVar()
input_field = Entry(frame2, text=input_user)
input_field.focus()

#创建一个 Button 控件 send_btn
send_btn = Button(frame2, text="发送", command=send_clicked)

#将 send_btn 和 input_field 都设置为向右对齐
send_btn.pack(side=RIGHT, padx=5, pady=5)
input_field.pack(side=RIGHT,expand=True, fill=BOTH)

#发送按钮点击事件处理函数，将 input_field 的输入内容添加到标签 messages 中
def send_clicked():
    input_get = input_field.get()
    messages['text'] = messages['text']+'\n'+input_get
    input_user.set('')
    #return "break"
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
window.mainloop()
```

程序运行界面如图 12-11 所示。

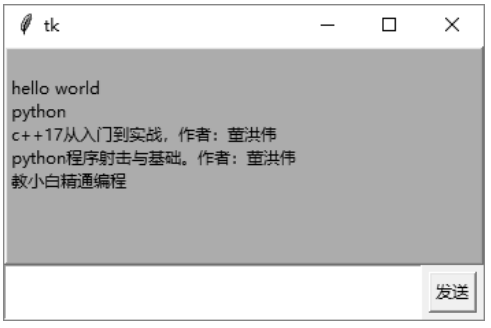


图 12-11　基于框架的聊天对话框界面

因为每次必须点击按钮才能发送消息，不如直接按 Enter(回车键)更加方便，所以可以通过自定义事件处理函数，即将输入文本框绑定到事件处理函数。例如：

```
input_field.bind("<Return>", Enter_pressed)
```

另外，对于多行文本，可以用 Text 代替 Label。但是，Text 默认可以编辑修改，所以可以通过在 insert()方法插入文本的前后开/关 Text 的编辑状态，使得这个 Text 只能显示文本而不能被用户修改。例如，

```
input_field.configure(state='normal')      #正常编辑状态
input_field.insert('end', 'Some Text')
input_field.configure(state='disabled')    #关闭编辑状态
```

下面是完整的程序：

```
from tkinter import *

window = Tk()
window.geometry('350x200')

#添加一个 frame
frame = Frame(window, relief=RAISED, borderwidth=2)

messages = Text(frame,state='disabled')
messages.pack(side=TOP, expand=1,fill=BOTH)

frame2 = Frame(window)
frame2.pack(side=BOTTOM,fill=X, expand=False)
frame.pack(fill=BOTH, expand=TRUE)


input_user = StringVar()
input_field = Entry(frame2, text=input_user)
input_field.focus()

def send_clicked():
    input_get = input_field.get()
    messages.configure(state='normal')
    messages.insert('end', input_get)
    messages.configure(state='disabled')
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        input_user.set('')

    def Enter_pressed(event):
        input_get = input_field.get()
        messages.configure(state='normal')
        messages.insert(INSERT, '%s\n' % input_get)
        messages.configure(state='disabled')
        input_user.set('')

    send_btn = Button(frame2, text="发送", command=send_clicked)
    send_btn.pack(side=RIGHT, padx=5, pady=5)
    input_field.pack(side=RIGHT,expand=True, fill=BOTH)

    input_field.bind("<Return>", Enter_pressed)

    window.mainloop()
```

然而，当输入的行数超过文本框的高度时，后面的输入无法显示出来，应该继续编写程序，使得输入的文本能够滚动。

一种方法是添加一个滚动条。例如：


```
    scrollbar = Scrollbar(frame)
    messages = Text(frame,state='disabled', yscrollcommand=scrollbar.set)

    scrollbar.config(command=messages.yview)
    scrollbar.pack(side="right", fill="y")
    messages.pack(side=TOP, expand=1,fill=BOTH)
```

更好的方法是利用 ScrolledText 代替 Text。例如：

```
    from tkinter import *
    from tkinter.scrolledtext import *

    window = Tk()
    window.geometry('350x200')

    #添加一个 frame
    frame = Frame(window, relief=RAISED, borderwidth=2)

    #scrollbar = Scrollbar(frame)
    #scrollbar.config(command=messages.yview)
    #scrollbar.pack(side="right", fill="y")
    #创建一个 ScrolledText 控件 messages
    messages = ScrolledText(frame,state='disabled')
    messages.pack(side=TOP, expand=1,fill=BOTH)

    frame2 = Frame(window)
    frame2.pack(side=BOTTOM,fill=X, expand=False)
    frame.pack(fill=BOTH, expand=TRUE)

    input_user = StringVar()
    input_field = Entry(frame2, text=input_user)
    input_field.focus()

    def send_clicked():
        input_get = input_field.get()
        messages.configure(state='normal')
        messages.insert(INSERT, '%s\n' % input_get)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        messages.configure(state='disabled')
        input_user.set('')
        messages.yview_moveto(1)                    #让 messages 滚动

    def Enter_pressed(event):
        input_get = input_field.get()
        messages.configure(state='normal')
        messages.insert(END,'%s\n' % input_get)
        #messages.insert(INSERT, '%s\n' % input_get)
        messages.configure(state='disabled')
        input_user.set('')

        messages.yview_moveto(1)                    #滚动
        #messages.yview(END)

    send_btn = Button(frame2, text="发送", command=send_clicked)

    send_btn.pack(side=RIGHT, padx=5, pady=5)
    input_field.pack(side=RIGHT,expand=True, fill=BOTH)

    input_field.bind("<Return>", Enter_pressed)

    window.mainloop()
```

如图 12-12 所示是基于 ScrolledText 的聊天框。

图 12-12　基于 ScrolledText 的聊天框

12.2　用类封装 GUI

12.2.1　菜单

首先看一个有菜单的应用程序：

```
from tkinter import *
from tkinter import filedialog

def NewFile():
    print("新建文件!")
def OpenFile():
    name = filedialog.askopenfilename()
    print(name)
def About():
    print("这是一个简单的菜单应用程序示例！")
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
root = Tk()
menu = Menu(root)
root.config(menu=menu)

filemenu = Menu(menu)
menu.add_cascade(label="File", menu=filemenu)
filemenu.add_command(label="New", command=NewFile)
filemenu.add_command(label="Open...", command=OpenFile)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=root.destroy)

helpmenu = Menu(menu)
menu.add_cascade(label="Help", menu=helpmenu)
helpmenu.add_command(label="About...", command=About)

root.mainloop()
```

该程序采用过程式编程思想，首先创建一个 Menu 对象，将根窗口(root)设置为其父控件，并配置为 root 的菜单：

```
menu = Menu(root)
root.config(menu=menu)
```


然后再创建两个 Menu 对象 filemenu 和 helpmenu，并将 Menu 对象作为它们的父控件(父菜单)，调用 Menu 的 add_cascade()方法将它们作为 Menu 菜单对象的子菜单。

```
filemenu = Menu(menu)
menu.add_cascade(label="File", menu=filemenu)

helpmenu = Menu(menu)
menu.add_cascade(label="Help", menu=helpmenu)
```

label="File"和 label="Help"是它们在 Menu 对象中显示的标签名。对于一个 Menu 对象，可以用 Menu 菜单类的 add_command()方法给它添加菜单项命令。例如：

```
filemenu.add_command(label="New", command=NewFile)
filemenu.add_command(label="Open...", command=OpenFile)
filemenu.add_command(label="Exit", command=root.destroy)
helpmenu.add_command(label="About...", command=About)
```

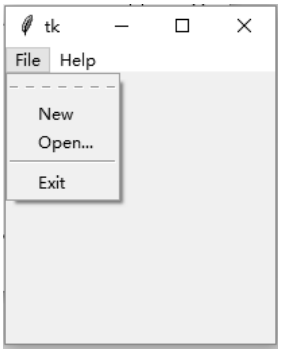


图 12-13　菜单程序用户界面

通过关键字参数 command 给每个菜单项设置相应的事情处理函数。例如：

```
command=NewFile
command=OpenFile
```

当用户选择该菜单项时，就会触发这个事情处理函数，执行菜单项对应的处理操作。

Menu 菜单类的 add_separator()方法可以给一个菜单添加分隔条，用于将菜单中的不同菜单项分隔为不同的分组。如图 12-13 所示为上述代码构建的用户界面。

基于面向对象编程思想，可以将用户界面的部件及其事件处理函数用一个类封装起来。例如，定义一个 MenuApp 类表示这个包含根窗口和菜单的应用程序：

```
from tkinter import Tk, Frame, Menu
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
class MenuApp:
    def __init__(self,master):              #master 是根窗口
        self.master = master
        master.title("First Menu App")

        menu = Menu(self.master)
        self.master.config(menu=menu)

        filemenu = Menu(menu)
        menu.add_cascade(label="File", menu=filemenu)

        filemenu.add_command(label="New", command=self.NewFile)
        filemenu.add_command(label="Open...", command=self.OpenFile)
        filemenu.add_separator()
        filemenu.add_command(label="Exit", command=self.onExit) #master.destroy)

        helpmenu = Menu(menu)
        menu.add_cascade(label="Help", menu=helpmenu)
        helpmenu.add_command(label="About...", command=self.About)


    def NewFile(self):
        print("新建文件!")

    def OpenFile(self):
        name = filedialog.askopenfilename()
        print(name)

    def About(self):
        print("这是面向对象的菜单应用程序示例！")

    def onExit(self):
        self.master.destroy() #self.master.quit()
```

MenuApp 类的 master 属性变量代表引用程序的根窗口，MenuApp 类的构造函数接收根窗口作为其参数，菜单项的命名处理函数设置为这个类的方法。

例如：

```
command=self.OpenFile
```

注意

作为方法，函数 OpenFile()的第一个参数必须是 self，即调用这个方法的那个对象自身的引用。

在下面的程序中将 Tk()返回的窗口传递给这个 MenuApp 类的构造函数：

```
def main():

    root = Tk()
    root.geometry("250x150+300+300")
    app = MenuApp(root)
    root.mainloop()
if __name__ == '__main__':
    main()
```

运行上述程序并执行各个菜单项，如 NewFile、About 和 OpenFile 等操作，产生如下的菜单项模拟操作结果：

```
新建文件!
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
这是面向对象的菜单应用程序示例!
C:/Users/s/Documents/desktop.ini
```

还可以让 MenuApp 类从一个顶层窗口，如从框架类 Frame 派生，从而可以自动继承 Frame 类的一些功能：

```
from tkinter import Frame, Tk, BOTH, Text, Menu, END
from tkinter import filedialog

class MenuApp2(Frame):
    def __init__(self):
        super().__init__()
        self.master.title("File dialog")
        self.pack(fill=BOTH, expand=1)

        menubar = Menu(self.master)
        self.master.config(menu=menubar)

        fileMenu = Menu(menubar)
        fileMenu.add_command(label="Open", command=self.onOpen)
        menubar.add_cascade(label="File", menu=fileMenu)

        self.txt = Text(self) #Text(self.master)
        self.txt.pack(fill=BOTH, expand=1)

    def onOpen(self):
        ftypes = [('Python files', '*.py'), ('All files', '*')]
        dlg = filedialog.Open(self, filetypes = ftypes)
        fl = dlg.show()

        if fl != '':
            text = self.readFile(fl)
            self.txt.insert(END, text)

    def readFile(self, filename):
        with open(filename, "r", encoding='utf-8') as f:
            text = f.read()
        return text

def main():

    root = Tk()
    ex = MenuApp2()
    root.geometry("300x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()
```


该程序有一个 file 菜单和 Text 文本框，在 filedialog.Open()方法中通过关键字参数 filetypes 可以设置打开文件的类型：

```
ftypes = [('Python files', '*.py'), ('All files', '*')]
dlg = filedialog.Open(self, filetypes = ftypes)
fl = dlg.show()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

如果 dlg 对话框返回的是一个非空的文件路径，则可调用 MenuExample 类的自身的 readFile()方法去读取这个文件的内容，并将内容插入 Text 文本框 self.txt 中。

如图 12-14 所示就是打开一个文本文件后读取的内容。

图 12-14　Text 部件中是读取文件的内容

12.2.2　工具条

和菜单一样，也可以给应用程序添加工具条，可以通过创建 Frame 类的对象来表示一个工具条：

```
        toolbar = Frame(self.master, bd=1, relief=RAISED)
```

然后在工具条上添加按钮 Button 对象，也可以给按钮对象设置一个图像，使得工具条看起来更直观。例如：

```
        open_img = Image.open("book_imgs/open-file.png").resize((36,36))
        openimg = ImageTk.PhotoImage(open_img)
        #在父控件 toolbar 中创建一个带图像的按钮 openButton
        openButton = Button(toolbar, image=openimg, relief=FLAT,command=self.onOpen)
        #也可在 Button 创建后设置其图像：openButton.image = openimg
        openButton.pack(side=LEFT, padx=2, pady=2)
        #将 toolbar 添加到父控件 self.master 中
        toolbar.pack(side=TOP, fill=X)
```

工具条的打开文件按钮的事件处理函数可以设置为 file 菜单的 open 菜单项的同一个事件处理函数。无论是通过菜单项还是按钮，都将触发同一个事件处理函数。例如：

```
        from tkinter import Frame, Tk, BOTH, Text, Menu, Button,END
        from tkinter import LEFT, TOP, X, FLAT, RAISED
        from tkinter import filedialog
        from PIL import Image, ImageTk

        class MenuApp3(Frame):
            def __init__(self):
                super().__init__()
                self.master.title("Tool Bar")
                self.pack(fill=BOTH, expand=1)

                menubar = Menu(self.master)
                self.master.config(menu=menubar)

                fileMenu = Menu(menubar)
                fileMenu.add_command(label="Open", command=self.onOpen)
                menubar.add_cascade(label="File", menu=fileMenu)

                #添加工具条
                toolbar = Frame(self.master, bd=2, relief=RAISED)

                #在父控件 toolbar 中创建一个带图像的按钮 openButton
                open_img = Image.open("book_imgs/open-file.png").resize((36,36))
                openimg = ImageTk.PhotoImage(open_img)
                openButton = Button(toolbar, image=openimg, relief=FLAT,command=self.onOpen)
                openButton.image = openimg

                openButton.pack(side=LEFT, padx=2, pady=2)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            toolbar.pack(side=TOP, fill=X)

            self.txt = Text(self.master)   #创建一个 Text 部件
            self.txt.pack(fill=BOTH, expand=1)
        def onOpen(self):
            ftypes = [('Python files', '*.py'), ('All files', '*')]
            dlg = filedialog.Open(self, filetypes = ftypes)
            fl = dlg.show()

            if fl != '':
                text = self.readFile(fl)
                self.txt.insert(END, text)

        def readFile(self, filename):
            with open(filename, "r", encoding='utf-8') as f:
                text = f.read()
            return text

    def main():
        root = Tk()
        ex = MenuApp3()
        root.geometry("300x250+300+300")
        root.mainloop()
    if __name__ == '__main__':
        main()
```


该程序的界面如图 12-15 所示。

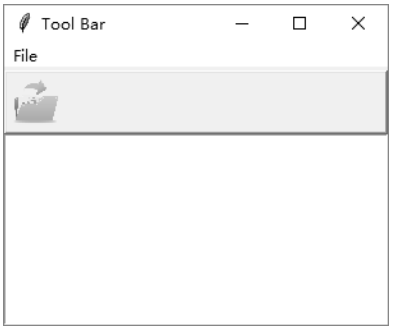


图 12-15 “工具条”界面

12.2.3 画图

Canvas 类是一个画布类，可以在屏幕上绘制各种图形。例如：

```
from tkinter import Tk, Canvas, Frame, BOTH
class CanvasExample(Frame):
    def __init__(self):
        super().__init__()

        self.master.title("Lines")
        self.pack(fill=BOTH, expand=1)

        canvas = Canvas(self)            #创建一个画布 canvas
        canvas.create_line(15, 25, 200, 25)
        canvas.create_line(300, 35, 300, 200, dash=(4, 2))
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        canvas.create_line(55, 85, 155, 85, 105, 180, 55, 85)

        #通过 outline 和 fill 设置边框和填充颜色
        canvas.create_rectangle(25, 30, 120, 70,outline="#f50", fill="#05f")

        canvas.pack(fill=BOTH, expand=1)


def main():

    root = Tk()
    ex = CanvasExample()
    root.geometry("400x250+300+300")
    root.mainloop()


if __name__ == '__main__':
    main()
```

除可以绘制线和矩形外，还可以绘制椭圆、多边形等其他图形。例如：

```
canvas.create_oval(10, 10, 80, 80, outline="gray", fill="gray", width=2)
                                        #绘制椭圆
canvas.create_arc(30, 200, 90, 100, start=0, extent=210, outline="gray",
    fill="gray", width=2)               #绘制弧

points = [150, 100, 200, 120, 240, 180, 210,
    200, 150, 150, 100, 200]
canvas.create_polygon(points, outline='gray', fill='gray', width=2)
                                        #绘制多边形
```

甚至还可以绘制一幅图像：

```
self.img = Image.open("myimage.jpg")
self.myimg = ImageTk.PhotoImage(self.img)

canvas = Canvas(self, width=self.img.size[0]+20,
         height=self.img.size[1]+20)
canvas.create_image(10, 10, anchor=NW, image=self.myimg)    #绘制一幅图像
```

或绘制文本：

```
canvas.create_text(20, 30, anchor=W, font="Purisa", text="Most relationships
                seem so transitory")
```

如图 12-16 所示，在画布上绘制了一条线段、一个矩形和一个三角形。

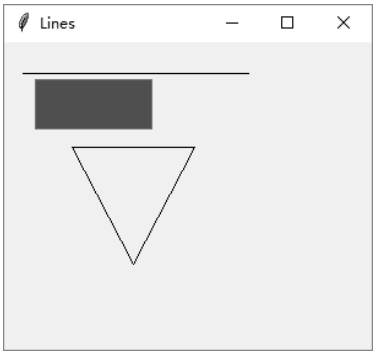


图 12-16　在 canvas 上绘制图形

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

12.2.4 用鼠标画图

对于鼠标事件，可以得到鼠标的位置 position。代码如下：

```
root = tk.Tk()
def motion(event):
  x, y = event.x, event.y
  print('{}, {}'.format(x, y))
root.bind('<Motion>', motion)
```

下面是一个基于橡皮条技术的画图程序，当用户按下鼠标左键拖曳时，将显示绘制的矩形形状，当放开鼠标时，就确定了一个矩形，如图 12-17 所示。

```
from tkinter import Tk, Canvas, Frame, BOTH
class DrawApp(Frame):
    def __init__(self):
        super().__init__()
        self.master.title("Drawing")
        self.pack(fill=BOTH, expand=1)

        self.canvas = Canvas(self)

        self.canvas.bind('<Button-1>', self.left_mouse_down)
        self.canvas.bind('<ButtonRelease-1>', self.left_mouse_up)
        self.canvas.bind('<B1-Motion>', self.left_mouse_motion)

        self.rects = []
        self.first = False                    #是否开始绘制
        self.draw_mode = 'rect'
        self.first_point = []                 #开始绘制的第一个起点
        self.canvas.pack(fill=BOTH, expand=1)

    def left_mouse_down(self,event):
        x, y = event.x, event.y
        #print('{}, {}'.format(x, y))
        self.first = not self.first           #改变 self.first 标志
        if self.first:
            self.first_point = [x,y]
        else:
            pass

    def left_mouse_up(self,event):
        self.first = not self.first
        pass

    def left_mouse_motion(self,event):
        x, y = event.x, event.y               #鼠标运动时的位置
        #print('{}, {}'.format(x, y))  #delete('all')
        if self.first:
            self.canvas.delete('moving_obj')   #删除之前绘制的 moving_obj
            self.canvas.create_rectangle(self.first_point[0], self.first_point[1],
                      x,y,tag="moving_obj")
def main():
    root = Tk()
    ex = DrawApp()
    root.geometry("300x200")
    root.mainloop()
if __name__ == '__main__':
    main()
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

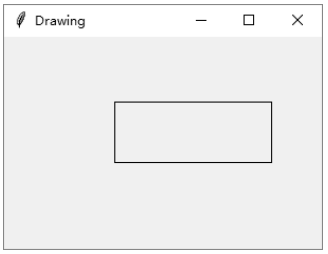


图 12-17　基于橡皮条技术的画图程序

作为练习，读者可以在以上程序的基础上编写一个类似二维 CAD 或 Windows 画图程序的应用程序，以绘制线段、多边形、圆弧等各种形状。当然，应用程序中应该有包含各种命令的菜单和工具条。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 13 章　网络套接字编程

计算机之间的网络通信需要在不同计算机程序之间发送和接收数据，通信的程序既可能位于同一个计算机上也可能位于不同的计算机上。和人们打电话或寄信一样，计算机程序之间的通信需要解决两个问题，通信双方的编址问题和协商收发数据的问题。

如同打电话是在两个电话号码间进行的，以及寄信时需要双方的地址一样，计算机程序之间的通信需要知道两个程序的地址，相互通信的互联网程序也需要有通信地址，网络程序的通信地址由计算机的主机地址(如互联网的 IP 地址)和通信端口构成。

除通信地址外，通信双方还需要协商如何发送和接收数据，如同网上购物的商家和买家从开始接触、下单、付款、发货、收货的过程一样，计算机程序之间的通信也必须遵守一套双方协商好的通信规则，如开始建立通信连接，相互发送和接收数据，终止连接等，这种指导双方通信的规则、标准称为**通信协议**。针对互联网的网络通信，人们发明了许多具有层次依赖关系的、满足不同通信需求的各种通信协议。如表 13-1 所示是 TCP 的层次协议栈及其含义。

表 13-1　TCP 的层次协议栈及其含义

TCP	含　义
应用层协议	为应用程序提供标准化数据交换，该层的协议包括超文本传输协议(HTTP)、文件传输协议(FTP)、邮局协议版本 3(POP3)、简单邮件传输协议(SMTP)和简单网络管理协议(SNMP)
传输层协议	负责维护整个网络的端到端通信，该层的协议包括传输控制协议(TCP)和用户数据报协议(UDP)。TCP 提供基于连接的主机之间的通信，并控制流量、多路复用和可靠性。UDP 提供基于数据报的通信
网络层协议	也称互联网层协议，处理数据包并连接独立网络，用于跨网络边界传输数据包。该层的协议包括 IP 和 Internet 控制消息协议(ICMP)，后者用于错误报告
物理层协议	用于链路上通信的协议，链路由网络中的节点或主机构成。该层的协议包括用于局域网(LAN)和以太网的地址解析协议(ARP)

Python 提供了两种编写网络通信程序的 API(库)。

- 低层的套接字库是底层操作系统中的 C 语言套接字(库)的 Python 接口，可以用来编写“面向连接”和“无连接”协议的客户和服务器程序。
- 高层的针对特定应用程序级别网络协议(如 FTP，HTTP 等)的库，它们依赖低层的套接字库。

13.1　套接字编程概述

套接字(socket)是双向通信通道的端点。应用程序通过“套接字”向网络发出请求或应答网络请求，使不同计算机上的应用程序进程可以相互通信。套接字可以在一个进程内，或者同一机器上的不同进程之间，以及不同网络的进程之间进行通信，并可以针对多种不同的通道类型(如 Unix 域套接字、TCP、UDP)实现套接字。套接字库提供了用于处理公共传输的特定类及相关的接口。

Python 的套接字模块是底层的，使用 BSD 套接字接口进行网络通信的 C 语言套接字 API 的 Python 接口，包含了套接字类 socket 处理实际的数据传输，以及网络相关的任务，如将服务器名转化为地址、格式化数据以便网络传输等。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

13.1.1 创建一个 socket 对象

套接字编程的第一步是首先使用 socket 模块的构造函数 socket()创建一个 socket 对象：

```
socket.socket([family[, type[, proto]]])
```

其中，

- family：套接字家族，可以是 AF_UNIX 或 AF_INET，通常是 AF_INET，以用于互联网的网络编程。
- type：套接字类型，可以根据是面向连接的网络通信还是面向无连接的网络通信分为 SOCKSTREAM 或 SOCKDGRAM。
- Protocol：默认为 0。

套接字类型分为 SOCKSTREAM 和 SOCKDGRAM，前者用于面向连接的网络通信，后者用于面向无连接的网络通信。而通信双方通常分别扮演两种角色，一种角色是服务器，用于监听来自其他套接字的网络请求；另一种角色是客户，用于发起网络请求，即向服务器请求网络服务。根据是否为面向连接的网络通信，套接字的网络通信主要分为 TCP 和 UDP 两种。

面向连接的 socket 通信要在通信双方之间建立一个可靠的、始终连接的通信管道，这个过程类似人们打电话的过程，数据发送和接收通过这个确定的连接进行：

```
import socket                                                  #导入 socket 模块
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)   #创建一个 socket 对象
```


无连接的 socket 通信无须通信双方建立连接，就可以直接根据对方的地址发送数据，发送的数据中包含了双方的地址，接收方可以从接收的数据中知道发送方的地址。这个过程类似人们发短信或邮寄信件：

```
import socket                                                  #导入 socket 模块
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)    #创建一个 socket 对象
```

13.1.2 服务器：绑定地址

无论是面向连接的网络通信还是面向无连接的网络通信的服务器程序，都需要首先将自己绑定到一个通信地址(如主机的 ip 地址和端口)：

```
host = socket.gethostname()          #获取本地主机名
port = 12345                         #设置端口
s.bind((host, port))                 #绑定端口
```

如果服务器和客户都位于(不属于互联网络的)同一台机器，则可以绑定到本机局部地址，即“127.0.01”或“localhost”：

```
host = 'localhost'
port = 12345
s.bind((host, port))
```

13.1.3 面向连接的监听

面向连接的服务器(TCP 服务器)在绑定地址后，可通过 listent()方法进入监听状态，等待来自客户的网络连接请求。一旦监听到来自客户的网络连接请求，就可以调用 accept()接收方法接收网络连接，该方法返回一个连接对象和客户的地址。之后就可以根据接收的网络连接中的客户地址进行双方的可靠通信。

注意

accept()接收方法返回的连接对象实际就是一个(关联另外一个端口)套接字。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

例如：

```
s.listen( )                    #等待客户端连接
c, addr = s.accept()           #建立客户端连接，accept()接收方法返回一个连接对象 connection
#和客户的地址
print('连接地址：',addr)
```

13.1.4 发送和接收数据

对于面向连接的 TCP 通信，双方调用套接字(socket)的 send()方法和 recv()方法发送和接收数据。例如：

```
c.send('hello')
c.recv(2048)                   #接收 2048 字节
c.close()                      #关闭连接
```

对于面向无连接的 TCP 通信，双方调用套接字(socket)的 sendto()方法和 recvfrom()方法发送和接收数据。例如：

```
s.recvfrom(1024)               #接收 1024 字节
s.sendto(bytes('hello', 'utf-8'), opposite_address)
                               #发送 bytes 字节序列到接收方地址 opposite_address
```


任何一方都可以关闭网络连接：

```
s.close()
```

13.2 TCP 服务器程序和客户程序

13.2.1 最简单的 TCP 服务器程序和客户程序

基于 TCP 的服务器和客户程序的双方通信过程如图 13-1 所示。

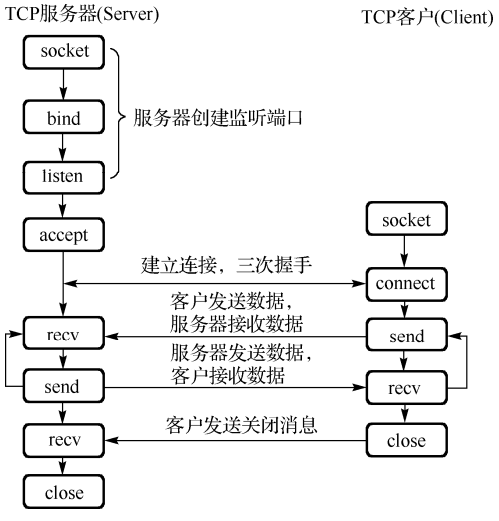


图 13-1 基于 TCP 的服务器和客户程序的双方通信过程

1. TCP 服务器程序

TCP 服务器程序(server.py)的主要步骤包括创建套接字对象、绑定本机地址、监听网络连接请求、接收网络连接请求、发送和接收数据、关闭连接。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

注意

发送和接收数据是通过建立连接后的连接对象进行的。

下面是 TCP 服务器程序 server.py 的完整代码:

```
import socket                    #导入 socket 模块
s = socket.socket()              #创建一个 socket 对象

host = socket.gethostname()      #获取本地主机名
port = 12345                     #设置端口
s.bind((host, port))             #绑定端口
s.listen( )                      #监听(等待)客户端口连接

c, addr = s.accept()             #与客户端口建立连接。accept()方法返回一个连接对象 connection
                                 #对象和客户的地址
print('连接地址: ', addr)

data = 'hello world'
c.send(data.encode('utf-8'))
c.close()                        #关闭连接
```

该程序将创建的 socket 对象绑定到一个网络主机和端口,并通过 listen()方法监听来自客户的网络连接请求，在 accept()方法接收一个网络连接请求后，通过返回的网络连接对象 c 发送一个字符串消息 “hello world”，然后关闭了该网络连接。

2．TCP 客户程序

TCP 客户程序(client.py)的主要步骤包括创建套接字对象、发送网络连接请求、发送和接收数据、关闭连接。

注意

TCP 客户不需要绑定地址，也不必监听和接收网络连接，而是直接使用 connect()方法发起连接请求。另外，发送和接收数据是通过套接字本身进行的。

下面是 TCP 客户程序 client.py 的完整代码：

```
#client.py
import socket                        #导入 socket 模块

s = socket.socket()                  #创建 socket 对象
host = socket.gethostname()          #获取本地主机名
port = 12345                         #设置端口

s.connect((host, port))
print(s.recv(1024).decode('utf_8'))
s.close()
```

该程序用创建的 socket 对象连接其他的网络主机和端口"s.connect((host, port))"，再通过 recv()方法接收数据(最多 1024 字节)。

3．运行 TCP 服务器程序和客户程序

可以打开两个终端，首先运行服务器等待客户连接：

```
>>>python server.py
```

运行客户程序，发送连接请求：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
>>>python client.py
```

在运行客户的终端窗口会看到服务器发来的信息：

```
hello
```

13.2.2 TCP 服务器程序和客户程序（多连接）

前面的 TCP 服务器程序和客户程序过于简单，仅是服务器发送/接收数据，客户接收/发送数据，并没有异常错误处理。并且，接收一个客户后，服务器程序就结束了，正常的服务器应该能够应付多个不同的客户。下面是改进的服务器程序和客户程序，服务器程序可以接收不同客户的连接，双方都能互相发送和接收数据，并且增加了异常错误处理程序。

1. TCP 客户程序

下面的客户程序（tcpclient2.py）在和服务器建立连接后，向服务器发送一份 UTF-8 编码的数据，然后接收服务器发回的数据并编码：

```
#tcp_client2.py
import socket
import sys

#创建一个 TCP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#将套接字(socket)连接到服务器正在监听的端口(port),即(服务器主机地址，端口)
server_host = 'localhost' #或服务器的 IP 地址
server_port = 12345
server_address = (server_host, server_port)
print('connecting to {} port {}'.format(*server_address))
sock.connect(server_address)
try:
    #发送数据
    data =input('请输入要发送的信息:')
    print("发送:    {}".format(data))
    sock.sendall(data.encode('utf-8'))

    #等待服务器的响应
    received = sock.recv(1024).decode('utf-8')
    print("接收的数据: {}".format(received))
finally:
    print('关闭套接字 socket')
    sock.close()
```

在 try 子句里处理发送和接收数据，无论是否出现异常，在 fmally 语句里都会关闭连接的套接字。

socket 模块有一个函数 create_connection()封装了套接字（socket）的创建和连接服务器的过程，可以用这个函数简化客户程序的代码：

```
#tcp_client2_.py
import socket
import sys

#创建一个 TCP socket，并连接到服务器
server_host = 'localhost'          #或服务器的 IP 地址
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
server_port = 12345
server_address = (server_host, server_port)
print('connecting to {} port {}'.format(*server_address))
sock = socket.create_connection(server_address)
try:
    #发送数据
    data =input('请输入要发送的信息:')
    print("发送:    {}".format(data))
    sock.sendall(data.encode('utf-8'))
    #等待服务器的响应
    received = sock.recv(1024).decode('utf-8')
    print("接收的数据: {}".format(received))
finally:
    print('关闭套接字 socket')
    sock.close()
```

2. TCP 服务器程序

下面的服务器程序(tcpserver2.py)在一个 while 循环里等待不同客户的连接，当建立一个客户连接后，又在一个嵌套的 while 循环里不断地“接收客户数据并将这份数据发回给客户”：

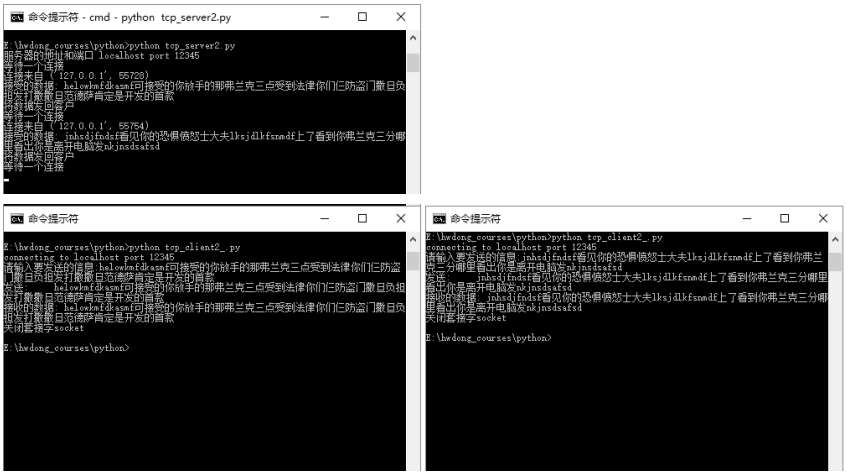

```
#tcp_server2.py
import socket
import sys

#创建一个 TCP 套接字(socket)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#绑定套接字(socket)到端口(port)
server_host = 'localhost'  #或服务器的 IP 地址,从而获得 server_host = socket.gethostname()
server_port = 12345
server_address = (server_host, server_port)
print('服务器的地址和端口 {} port {}'.format(*server_address))
sock.bind(server_address)

#监听进来的连接请求
sock.listen( )
while True:
    #等待一个连接
    print('等待一个连接')
    connection, client_address = sock.accept()
    try:
        print('连接来自', client_address)
        #以小块的方式接收数据并发送回
        while True:
            try:
                data = connection.recv(1024).decode('utf_8')
                if not data: break
                print("接收的数据: {}".format(data))
                print('将数据发回客户')
                connection.sendall(data.encode('utf_8'))
            except:
                break
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        finally:
            #关闭连接
            connection.close()
```

3．运行 TCP 服务器程序和客户程序

打开一个终端窗口运行 tcpserver2.py，再打开多个窗口运行 tcpclient2.py。如图 13-2 所示是运行多连接 TCP 服务器程序和客户程序的截图。


命令提示符 - cmd - python tcp_server2.py
E:\hwdong_courses\python>python tcp_server2.py
服务器的地址和端口 localhost port 12345
等待一个连接
连接来自 ('127.0.0.1', 55728)
接收的数据: helowkmfdksamf可接受的你缺手的那弗兰克三点要到法律你们三防盗门歉且负担责打歉歉且范德萨肯定是开发的首款
发送数据给客户
等待一个连接
连接来自 ('127.0.0.1', 55754)
接收的数据: jnhsdjfndsf看见你的恐惧愤怒士大夫lksjdlkfsnmdf上了看到你弗兰克三分哪里看出你是离开电脑发nkjnsdsafsd
发送数据给客户
等待一个连接
命令提示符
E:\hwdong_courses\python>python tcp_client2_.py
connecting to localhost port 12345
请输入要发送的信息:helowkmfdksamf可接受的你缺手的那弗兰克三点要到法律你们三防盗门歉且负担责打歉歉且范德萨肯定是开发的首款
发送: helowkmfdksamf可接受的你缺手的那弗兰克三点要到法律你们三防盗门歉且负担责打歉歉且范德萨肯定是开发的首款
接收的数据: helowkmfdksamf可接受的你缺手的那弗兰克三点要到法律你们三防盗门歉且负担责打歉歉且范德萨肯定是开发的首款
关闭套接字socket
E:\hwdong_courses\python>
命令提示符
E:\hwdong_courses\python>python tcp_client2_.py
connecting to localhost port 12345
请输入要发送的信息:jnhsdjfndsf看见你的恐惧愤怒士大夫lksjdlkfsnmdf上了看到你弗兰克三分哪里看出你是离开电脑发nkjnsdsafsd
发送: jnhsdjfndsf看见你的恐惧愤怒士大夫lksjdlkfsnmdf上了看到你弗兰克三分哪里看出你是离开电脑发nkjnsdsafsd
接收的数据: jnhsdjfndsf看见你的恐惧愤怒士大夫lksjdlkfsnmdf上了看到你弗兰克三分哪里看出你是离开电脑发nkjnsdsafsd
关闭套接字socket
E:\hwdong_courses\python>
```

图 13-2　多连接 TCP 服务器程序和客户程序



### 13.2.3　TCP 服务器程序和客户程序(数据分块)

从上面的运行过程(如图 13-2 所示)可以看出，当数据很大时，一次接收整个数据是不可行的，通常的方法是，将数据分成固定大小的块(chunks)发送。因为数据是分块接收到的，所以可以编写一个辅助函数，用于将这些数据分块组合成完整的数据：

```
def recvall(sock, buffer_size=4096):
    buf = sock.recv(buffer_size)        #每次接收大小为 buffer_size 的数据块
    while buf:
        yield buf
        if len(buf) < buffer_size: break #最后的数据块小于 buffer_size
        buf = sock.recv(buffer_size)
   return buf
```

当最后一个数据块的大小小于 buffer_size 时，说明已经接收到完整的数据了。

下面分别是改进后的客户程序和服务器程序。

1．TCP 客户程序

在 TCP 客户程序(tcp_client_chunks.py)中，客户通过套接字的 sendall()方法发送数据，并通过自定义的 recvall()方法接收数据块。recvall()方法的参数 buffer_size 被故意设置为 16 字节，以验证分块接收数据的过程。接收的数据被逐一追加到表示完整数据的 data 中：

```
#tcp_client_chunks.py
import socket
import sys


def recvall(sock, buffer_size=4096):
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
```

```
        buf = sock.recv(buffer_size)
        while buf:
            yield buf
            if len(buf) < buffer_size: break
            buf = sock.recv(buffer_size)

    #创建一个 TCP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    #将套接字(socket)连接到服务器正在监听的端口(port),即(服务器主机地址，端口)
    server_host = 'localhost' #或服务器的 IP 地址
    server_port = 12345
    server_address = (server_host, server_port)    #服务器通信地址
    print('connecting to {} port {}'.format(*server_address))
    sock.connect(server_address)                   #连接服务器地址
    try:
        #发送数据
        data =input('请输入要发送的信息:')
        print("Sent:     {}".format(data))
        sock.sendall(data.encode("utf-8"))
        #等待服务器的响应
        data = b''
        for buf in  recvall(sock, 16):
            data += buf;

        print("接收的数据: {}".format(data.decode("utf-8")))
    finally:
        print('关闭套接字 socket')
    sock.close()
```

2．TCP 服务器程序

在 TCP 服务器程序(tcp_server_chunks.py)中，服务器通过自定义的 recvall()方法接收客服的数据，传递给 recvall()方法的参数 buffer_size 同样为 16 字节。接收的这些数据库被组成 data，通过连接对象 connection 的 sendall()方法发送回客户。

```
    #tcp_server_chunks.py
    import socket
    import sys

    def recvall(sock, buffer_size=4096):
        buf = sock.recv(buffer_size)
        while buf:
            yield buf
            if len(buf) < buffer_size: break
            buf = sock.recv(buffer_size)

    #创建一个 TCP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    #绑定套接字到 port(端口)
    server_host = 'localhost'   #或服务器的 IP 地址,从而获得 server_host = socket.gethostname()
    server_port = 12345
    server_address = (server_host, server_port)
    print('服务器的地址和端口 {} port {}'.format(*server_address))
    sock.bind(server_address)

    #监听连接请求
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        sock.listen(1)

        while True:
            #等待一个连接
            print('等待一个连接')
            connection, client_address = sock.accept()
            try:
                print('连接来自', client_address)
                data = b''
                for buf in  recvall(connection, 16):
                    data += buf;

                print("接收的数据：{}".format(data.decode("UTF-8")))
                print('将数据发回客户')
                connection.sendall(data)

            finally:
                #关闭连接
                connection.close()
        sock.close()
```

3．运行 TCP 服务器程序和客户程序

如图 13-3 所示为分块接收数据的 TCP 服务器程序和客户程序运行截图，用于保证接收完整的数据。对于发送数据也可以同样处理。读者可作为练习。

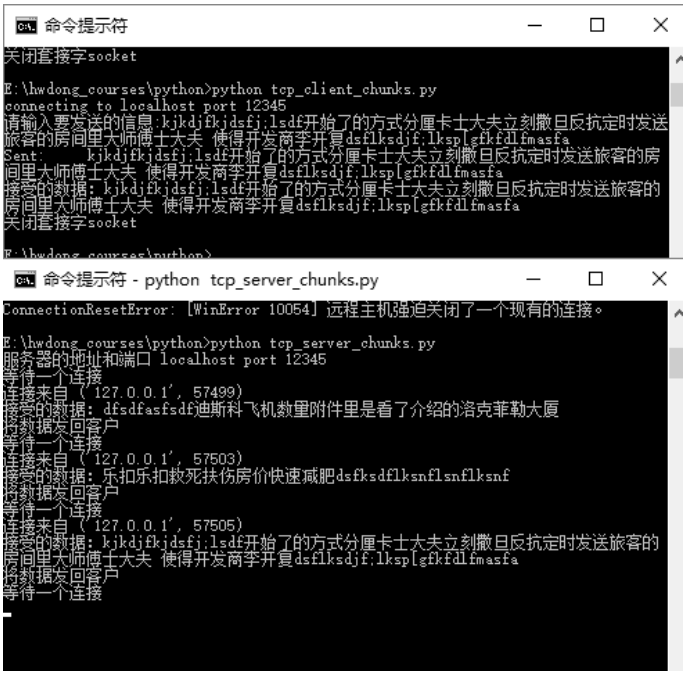


图 13-3　分块接收数据的 TCP 服务器程序和客户程序

13.2.4　TCP 服务器程序（多进程）

上面的 TCP 服务器程序每次只能服务于一个客户，即当一个客户在对话时，其他客户端无法连接。由此可知，服务器是轮流处理每个客户的。更好的方式是让服务器可以并发（同时）处理多个客户连接。可以通过多进程模块 multiprocessing 的函数 process()为每个客户创建一个专门的进程：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
#tcp_server_process.py
import socket, sys, os, multiprocessing
def worker(connection, client_address):
    try:
        print('连接来自', client_address)
        #以小块的方式接收数据并发送回客户
        chunk_size= 16   #chunk_size= 1024
        while True:
            try:
                data = connection.recv(chunk_size)
                if not data: break
                print("接收的数据: {}".format(data)) #.decode("utf-8")))
                print('将数据发送回客户')
                connection.sendall(data)
            except:
                break
    finally:
        connection.close()

if __name__ == "__main__":
  #创建一个 TCP 套接字
  sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

  #绑定套接字到 port(端口)
  server_host = '0.0.0.0'   #或服务器的 IP 地址,从而获得 server_host = socket.gethostname()
  server_port = 12345
  server_address = (server_host, server_port)
  print('服务器的地址和端口 {} port {}'.format(*server_address))
  sock.bind(server_address)

  #监听连接请求
  sock.listen(1)
  while True:
     print('等待一个连接')
     connection, client_address = sock.accept()
     process = multiprocessing.Process(target=worker, args=(connection, client_address,))
     process.daemon = True
     process.start()

  sock.close()
```

13.2.5 TCP 服务器程序(多线程)

为每个客户连接单独创建一个专门的进程会消耗很多内存资源，当客户连接很多时，这是无法想象的。因此，更好的办法是用“轻量级的进程”，即“线程”代替进程，即为每个客户创建一个单独的线程(单线程)。可以通过多线程模块 threading 的 Thread 类创建一个 Thread 线程对象，创建这个线程对象时需要传递一个线程处理函数 handler()，并调用 start()方法启动这个线程：

```
#tcp_server_thread.py
import socket,sys, os, threading

#生成一个 TCP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#绑字套接字到 port(端口)
server_host = 'localhost'   #或服务器的 IP 地址,从而获得 server_host = socket.gethostname()
server_port = 12345
server_address = (server_host, server_port)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print('服务器的地址和端口 {} port {}'.format(*server_address))
sock.bind(server_address)

#监听连接请求
sock.listen(1)
while True:
    print('等待一个连接')
    connection, client_address = sock.accept()
    def handler():
        try:
            print('连接来自', client_address)
            #以小块的方式接收数据并发送回客户
            chunk_size= 16    #chunk_size= 1024
            while True:
                try:
                    data = connection.recv(chunk_size)
                    if not data: break
                    print("接收的数据: {}".format(data))
                    print('将数据发送回客户')
                    connection.sendall(data)
                except:
                    break

        finally:
          #关闭连接
          connection.close()

    threading.Thread(target=handler).start()

sock.close()
```


除处理客户数据请求并响应外，服务器还可以通过另外一个线程(线程处理函数 notify())向同一个客户连接推送信息，如新产品信息等：

```
#tcp server thread notify.py
import socket,sys,os, time, threading

#生成一个 TCP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#绑定套接字到 port(端口)
server_host = 'localhost'   #或服务器的 IP 地址,从而获得 server_host = socket.gethostname()
server port = 12345
server address = (server host, server port)
print('服务器的地址和端口 {} port {}'.format(*server_address))
sock.bind(server address)

#监听连接请求
sock.listen(1)
while True:
    print('等待一个连接')
    connection, client address = sock.accept()
    def handler():
        try:
            print('连接来自', client address)
            #以小块的方式接收数据并发送回客户
            chunk_size= 16    #chunk_size= 1024
            while True:
                try:
                    data = connection.recv(chunk_size)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                    if not data: break
                    print("接收的数据: {}".format(data))
                    print('将数据发送回客户')
                    connection.sendall(data)
                except:
                    break
            finally:
              #关闭连接
              connection.close()

        def notify():
            try:
                while True:
                    time.sleep(5)
                    connection.sendall('有新文章'.encode("utf-8"))
            except:
               pass
            finally:
              #关闭连接
              connection.close()

        threading.Thread(target=handler).start()
        threading.Thread(target=notify).start()
    sock.close()
```


13.3 UDP 服务器程序和客户程序

用户数据报协议(User Datagram Protoco，UDP)与 TCP 的工作方式不同。TCP 是面向流的协议，能够确保所有数据以正确的顺序传输。UDP 是面向消息的协议，不需要建立连接，因此 UDP 套接字编程更简单。

UDP 通信类似日常生活中的邮寄信息、发送短信、邮寄包裹，UDP 消息是以数据报形式发送的，每个数据报中包含发送和接收方的地址，每个数据报的大小是有限制的(对于 IPv4，这意味着它们只能保存 65 507 字节，因为 65 535 字节的数据报中也包含标题信息)。如果数据超过数据报的大小，则需要将数据分割成多个数据报单独发送，而接收方需要将不同时间接收的无序数据报组装成原始的数据。

基于 UDP 的服务器和客户程序的双方通信过程如图 13-4 所示。

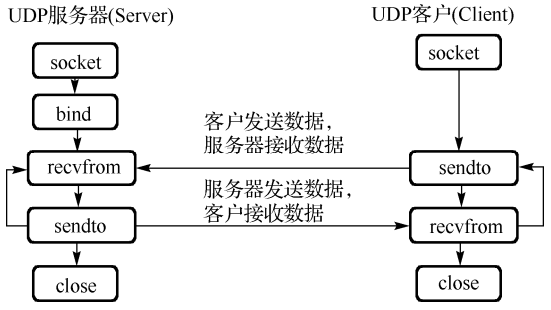


图 13-4　基于 UDP 的服务器和客户程序的双方通信过程

13.3.1 UDP 服务器程序

UDP 服务器程序(udp_server.py)的主要步骤包括创建套接字对象、绑定本机地址、接收和发送数据、关闭连接：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
#udp_server.py
import socket
import time

host = '127.0.0.1'  #或 host = 'localhost'
port = 12345
server_address = (host, port)

#创建一个套接字对象
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#绑定套接字到 port(端口)
sock.bind(server_address)   #sock.bind(('localhost', 23456))
while True:
    try:
        print('等待消息...')
        data, addr = sock.recvfrom(4096)
        data = data.decode('utf-8')
        print('消息来自', addr)
        print('收到的消息:', data)
        sock.sendto(data.encode('utf-8'), addr)  #bytes(data, 'UTF-8'), addr)
    except:
        break

sock.close()
```


13.3.2 UDP 客户程序

和 UDP 服务器的区别是 UDP 客户不需要绑定地址。

UDP 客户程序(udp_client.py)的主要步骤包括创建套接字对象、接收和发送数据、关闭连接：

```
#udp_client.py
import socket

host = '127.0.0.1'  #或 host = 'localhost'
port = 12345
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_address = (host, port)      #服务器地址，包括端口号

while True:
    try:
        msg = input('输入发送消息: ')
        if not msg:
            break
        #sock.sendto(bytes(msg, 'utf-8'), server_address) #该函数返回发送出去的字节数
        sock.sendto(msg.encode('utf-8'), server_address)  #该函数返回发送出去的字节数
        data, addr = sock.recvfrom(4096)
        data = data.decode('utf-8')
        print('相应的消息:', data)
    except:
        break

sock.close()
```

如图 13-5 所示是运行 UDP 服务器程序和客户程序的截图。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

图 13-5　UDP 服务器程序和客户程序

13.4　socketserver

13.4.1　socketserver 模块

Python 在 socket 模块基础上提供了更方便的 socketserver 模块，可以使网络服务器的编写更加简单。

该模块一共有四个基本的服务器类：

- socketserver.TCPServer(serveraddress, RequestHandlerClass, bindand_activate=True)
- socketserver.UDPServer(serveraddress, RequestHandlerClass, bindand_activate=True)
- socketserver.UnixStreamServer(serveraddress, RequestHandlerClass, bindand_activate=True)
- socketserver.UnixDatagramServer(serveraddress, RequestHandlerClass, bindand_activate=True)

其中，后两个类用于 Unix 系统(使用的是 Unix 域的 socket)。

这四个类都用于同步处理网络请求(只有一个请求处理完，才能接着处理下一个请求)。如果一个请求需要很长时间才能完成，则应创建单独的进程或线程来处理它。可以用 ForkingMixIn 和 ThreadingMixIn 这两个类支持异步行为。

创建一个服务器的步骤如下。

(1) 必须通过继承 BaseRequestHandler 类并覆盖其 handle()方法来创建“请求处理器类”。handle()方法将处理传入的请求。

(2) 必须实例化一个服务器类，并向其传递服务器的地址和请求处理程序类。建议在 with 语句中使用服务器。

(3) 调用服务器对象的 handle_request()方法或 server_forever()方法来处理一个或多个请求。

(4) 调用 server_close()方法来关闭套接字(除非已使用了 with 语句)。

13.4.2　socketserver.TCPServer

下面是一个简单的基于 TCPServer 服务器的程序代码：

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    def handle(self):
        #self.request 是连接到客户端的 TCP 套接字
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        #将同样的数据转换成大写字母发送回客户
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        #创建服务器，绑定到 localhost 上的端口 9999
        with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
            #Activate the server; this will keep running until you
            #interrupt the program with Ctrl-C
            server.serve_forever()
```

对于面向数据流的连接(如 TCP)，可使用另一个请求处理器类(socketserver.StreamRequestHandler)定义“请求处理器类”，它提供标准文件接口来简化通信的过程：

```
    class MyTCPHandler(socketserver.StreamRequestHandler):
        def handle(self):
            #self.rfile 是由处理器创建的类似文件的对象
            #可以使用 readline()代替 recv()
            self.data = self.rfile.readline().strip()
            print("{} 写数据:".format(self.client_address[0]))
            print(self.data)
            #同样，self.wfile 是一个类似文件的对象，用于回写数据到客户端
            self.wfile.write(self.data.upper())
```

两个处理程序不同之处在于第二个处理程序中的 readline()会多次调用 recv()，直到它遇到换行符为止，而第一个处理程序中的单个 recv()调用将仅返回从客户发送的一个 sendall()调用的数据。

客户程序代码仍然和之前的是一样的：

```
    import socket
    import sys
    HOST, PORT = "localhost", 9999
    data = input('输入发送消息: ')   #" ".join(sys.argv[1:])      #来自命令行的输入

    #创建一个套接字(socket)对象 (SOCK_STREAM 意思是 TCP socket)
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        #连接到服务器并发送数据
        sock.connect((HOST, PORT))
        sock.sendall(bytes(data + "\n", "utf-8"))
        #从服务器接收数据，然后关闭连接 (with 语句会自动关闭连接)
        received = str(sock.recv(1024), "utf-8")

    print("发送数据:     {}".format(data))
    print("接收的数据: {}".format(received))
```

13.4.3 socketserver.UDPServer

以下是一个简单的、基于 UDPServer 服务器的程序代码：

```
    import socketserver
    class MyUDPHandler(socketserver.BaseRequestHandler):

        def handle(self):
            data = self.request[0].strip()
            socket = self.request[1]
            print("{} wrote:".format(self.client_address[0]))
            print(data)
            socket.sendto(data.upper(), self.client_address)

    if __name__ == "__main__":
        HOST, PORT = "localhost", 9999
        with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
            server.serve_forever()
```

客户程序代码仍然和以前的一样：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import socket
import sys

HOST, PORT = "localhost", 9999
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
data = input('请输入发送数据：')

#UDP 不需要建立连接.
#相反，数据通过 sendto()直接发送给接收方
#sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
sock.sendto(data.encode("utf-8"), (HOST, PORT))
received = sock.recv(1024).decode("utf-8")

print("发送数据:    {}".format(data))
print("接收的数据:  {}".format(received))
```

总结

本章介绍套接字编程，包括如下内容。

- 套接字编程的基本概念和 Python 的套接字模块 socket。
- 编写面向连接的 TCP 服务器和客户程序。
- 轮流处理请求的服务器。
- 服务器用多进程和多线程处理并发请求。
- 编写无连接的 UDP 服务器和客户程序。
- 用更高层的模块 socketserver 中的 socket 服务器类，如 TCPServer 和 UDPServer，编写套接字服务器。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 14 章　Internet 应用编程

Python 提供多个可以访问 Internet 和处理网络协议的模块(包)，这些模块(包)中的大多数都需要依赖系统的 socket 模块。

http 模块包含多个处理 HTTP(HyperText Transfer Protocol，超文本传输协议)的模块(如 http.client、http.server、http.cookies、http.cookiejar 等)。HTTP 是对浏览器和服务器之间发送和响应(基于 URL 的)资源请求和结果的网络协议(规则)，浏览器向服务器发送一个 GET 或 POST 请求，服务器返回该请求的结果。

urllib 模块是基于 http 模块的高层接口，提供基于 HTTP 下载 URL 数据的功能，URL(统一资源定位符)是互联网上符合 URL 格式标准的资源的地址。可以用 URL 表示一个网页、图片、视频、文件等的地址，如最常见的是在浏览器里输入的网址。

smtplib 模块提供基于 SMTP(简单邮件传输协议)发送 email 的功能。smtplib 模块定义了一个 SMTP 客户端会话对象，可将邮件发送到任何运行 SMTP 或 ESMTP 监听守护程序的 Internet 计算机。

poplib 模块提供基于 POP3(邮局协议第 3 版)的客户端功能，该模块定义了一个 POP3 类，它封装了与 POP3 服务器的连接，并实现了 RFC 1939 中定义的协议，用于收取邮件。

imaplib 模块提供 IMAP4(因特网信息访问协议)客户端，是用于接收邮件的协议。该模块定义了三个类，IMAP4、IMAP4_SSL 和 IMAP4_stream，它们封装了与 IMAP4 服务器的连接，并实现了 RFC 2060 中定义的 IMAP4rev1 客户端协议的大部分。

更多 Internet 应用模块介绍请参考https://docs.python.org/3/library/internet.html。

下面仅以打开 URL 资源的 urllib 模块和发送接收邮件的 smtplib、poplib、imaplib 模块对 Internet 应用编程做一个简单的介绍。

14.1　urllib 模块

urllib 提供了一系列用于操作 URL 的功能。例如，urllib 的 request 模块可以通过 urlopen 下载 URL 内容。

14.1.1　Get 请求

向函数 urlopen()传递一个 URL 网址，即发送一个 Get 请求，并返回一个响应对象：

```
import urllib.request
response = urllib.request.urlopen('http://python.org/')
html = response.read()
```

变量 html 执行的对象中包含了返回相应的 html 网页的内容。函数 urlopen()根据输入的 URL 的类型不同返回不同类型的响应对象，返回的对象通常提供以下方法：

- info()：返回 URL 相关的元数据。
- getcode()：返回 http 状态码。如果 URL 不是 http URL，则返回 None。
- geturl()：返回请求的 URL。
- 对 HTTPResponse 类型数据进行操作的方法，如 read()、readline()、readlines()、fileno()、close()等。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

函数 urlopen()返回的响应信息 response 的 response.headers 包含了和 info()返回的响应信息一样的元数据（称为响应头）。例如：

```
        #print("headers:",response.headers)
        print("info:",response.info())
```

输出：

```
        info: Server: nginx
        Content-Type: text/html; charset=utf-8
        X-Frame-Options: DENY
        Via: 1.1 vegur
        Via: 1.1 varnish
        Content-Length: 48903
        Accept-Ranges: bytes
        Date: Sat, 07 Sep 2019 03:20:19 GMT
        …
```

response 是一个可迭代对象，可以迭代地访问相应内容的每一行。例如：

```
        with urllib.request.urlopen('http://python.org/') as response:
           print(response.headers)                          #打印响应头
           for line in response:
              line = line.decode('utf-8')                   #将字节流解码为字符串
              if 'Events' in line or 'News' in line:        #查找包含关键字的行
                  print(line)
```

上述代码输出的 URL 为“http://python.org/”的响应头和包含关键字 Events 或 News 的行。

```
        <link  rel="alternate"  type="application/rss+xml"  title="Python  Software
Foundation News"
href="https://feeds.feedburner.com/PythonSoftwareFoundationNews">
              <a  href="/blogs/"  title="News  from  around  the  Python  world"
class="">News</a>
        …
```

有些网页访问请求需要携带一些数据，如在搜索引擎中输入的搜索关键字。可以将查询串添加到 URL 网址后面。为此，首先要将查询串编码成服务器可以识别的字符串（如编码为 UTF-8 字符串），需要用 urllib.parse 模块中的函数 urlencode()对数据编码。

例如，下列代码用关键字'hwdong'和编码 utf-8 查询百度网上的信息：

```
        url =  "https://www.baidu.com/s?"
        params =  {'wd': 'hwdong', 'ie': 'utf-8'}
        query_string = urllib.parse.urlencode( params )
        print(url+query_string)
        with urllib.request.urlopen( url+query_string ) as response:
           response_text = response.read()
           print( response_text )
```

其中，查询参数'wd'表示关键字，而'ie'表示编码。

14.1.2 Post 请求

Get 请求用于请求服务器的资源，Post 请求向服务器发送信息同于创建和更新服务器数据，如发送用户注册信息。发送 Post 请求前，同样要将发送的数据编码成服务器可以识别的字符串，即用 urllib.parse 模块中的函数 urlencode()对数据编码。和 Get 请求不同的是，Post 请求发送的数据是作为单独的参数传递给 urlopen()的。例如：

```
        import urllib.parse
        import urllib.request
        url = 'https://www.baidu.com/s?'
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
values = {'wd': 'hwdong', 'ie': 'utf-8'}
data = urllib.parse.urlencode(values)
data = data.encode('utf-8') # data should be bytes
with urllib.request.urlopen(url, data) as response:
   response_text = response.read()
   print( response_text )
```

14.1.3 Request 对象

可以将 URL 和要发送的数据通过函数 urllib.request.Request() 包裹成一个 Request 对象传递给 urlopen() 等函数。

构造一个 Request 对象：

```
urllib.request.Request(url, data=None, headers={}, origin_req_host=None,
unverifiable=False, method=None)
```

其中，url 是必选参数，其他都是可选参数；data 是用户自己的数据；headers 表示 HTTP 头；origin_req_host 是请求者的主机地址；method 表示请求的方式，如 Get、Post、Put、Delete 等。

对于上面的 Post 请求，也可以采用如下的代码：

```
import urllib.parse
import urllib.request
url = 'https://www.baidu.com/s?'
values = {'wd': 'hwdong', 'ie': 'utf-8'}
data = urllib.parse.urlencode(values)
data = data.encode('utf-8') # data should be bytes
req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response:
   response_text = response.read()
   print( response_text )
```


有些网站为防止他人恶意采集其信息而进行了一些反爬虫的设置，为了能够下载这些 URL 资源，可以在发送的请求中给 HTTP 头添加一些信息，从而将请求伪装成浏览器（如模拟 iPhone 8）去请求相应的资源。例如，下面的代码发送一个伪装成浏览器请求的请求。

```
import urllib.request
url = "http://pyton.org/"
headers = {
   'User-Agent': 'Mozilla/8.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X) AppleWebKit/
   536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25'
}
request = urllib.request.Request(url=url, headers=headers)
response = urllib.request.urlopen(request)
print(response.read().decode('utf-8'))
```

也可以首先用 URL 构造一个 Request 对象，然后调用 Request 的方法添加其他信息。例如：

```
from urllib import request
req = request.Request('http://python.org/')
req.add_header('User-Agent', 'Mozilla/8.0 (iPhone; CPU iPhone OS 8_0 like Mac
OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e
Safari/8536.25')
with request.urlopen(req) as f:
   print('Status:', f.status, f.reason)
   for k, v in f.getheaders():
      print('%s: %s' % (k, v))
   print('Data:', f.read().decode('utf-8'))
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

下面是一个简单的登录微博的代码：

```
from urllib import request, parse
username = input('输入用用户名如手机号: ')
passwd = input('输入密码: ')
login_data = parse.urlencode([
    ('username', username),
    ('password', passwd),
    ('entry', 'mweibo'),
    ('client_id', ''),
    ('savestate', '1'),
    ('ec', ''),
    ('pagerefer', 'https://passport.weibo.cn/signin/welcome?entry=mweibo&r=
    http%3A%2F%2Fm.weibo.cn%2F')
])

req = request.Request('https://passport.weibo.cn/sso/login')
req.add_header('Origin', 'https://passport.weibo.cn')
req.add_header('User-Agent', 'Mozilla/8.0 (iPhone; CPU iPhone OS 8_0 like Mac
OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e
Safari/8536.25')
req.add_header('Referer', 'https://passport.weibo.cn/signin/login?entry=
mweibo&res=wel&wm=3349&r=http%3A%2F%2Fm.weibo.cn%2F')

with request.urlopen(req, data=login_data.encode('utf-8')) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))
```

返回的 retcode 的值表示登录是否成功，如“retcode:50011015”表示登录失败。不同网站的登录验证的过程是不同的，如何登录微博等网站请读者网上搜索相关教程。

14.1.4 代理服务器

如果用同一 IP 爬取同一个网站上的网页，则会被该网站服务器屏蔽。解决方法是，通过代理服务器请求网站服务器的资源。为了使用代理，要用到 opener 和 handler 这两个类。具体地讲就是，用一个从 handler 派生出的 ProxyHandler 类的对象构造一个 opener 对象，并安装这个 opener 对象：

```
import urllib.request
url = "http://python.org/"
headers = {
    'User-Agent': 'Mozilla/8.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X)
    AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e
    Safari/8536.25'
}

#创建一个包含代理信息的 proxy_handler 对象
proxy_handler = urllib.request.ProxyHandler({
    'http': 'web-proxy.com:8080',
    'https': 'web-proxy.com:8080'
})

opener = urllib.request.build_opener(proxy_handler)     #构造一个 opener 对象
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
urllib.request.install_opener(opener)                    #安装 opener 对象

request = urllib.request.Request(url=url, headers=headers)
response = urllib.request.urlopen(request)
print(response.read().decode('utf-8'))
```

注意

需要使用一个可用的代理服务器。

14.1.5 登录验证

有的网站需要登录验证后才能访问，因此需要创建一个 HTTPPasswordMgrWithDefaultRealm 类的对象，再用这个对象构建一个 HTTPBasicAuthHandler 对象，然后用函数 build_opener()创建一个 opener 对象，最后用 opener 的函数 open()发起请求：

```
import urllib.request
url = "http://tieba.baidu.com/"
user = 'user'
password = 'password'
pwdmgr = urllib.request.HTTPPasswordMgrWithDefaultRealm()
pwdmgr.add_password(None, url, user, password)

auth_handler = urllib.request.HTTPBasicAuthHandler(pwdmgr)
opener = urllib.request.build_opener(auth_handler)
response = opener.open(url)
print(response.read().decode('utf-8'))
```


14.1.6 网络爬虫

所谓的“网络爬虫”程序是通过 urllib 抓取 URL 网页的功能。如同在浏览器输入 URL 网址打开网页一样，爬虫程序也是通过发送 Get 或 Post 请求下载这些数据，然后对下载的网页内容进行分析，如通过 re 模块的正则表达式模式匹配，提取出自己感兴趣的内容。因为网页内容里有其他的 URL 链接，所以可以继续下载这些 URL 链接中的新的数据，这样可以不断地下载自己感兴趣的内容。

下面就是一个简单的爬虫程序，可以爬取百度网页中的图片。运行这个程序时，输入要爬取的图片的关键字，如“动漫”，程序将根据百度搜索的结果提取出图片的链接，然后下载这些图片：

```
from urllib.request import urlopen
from urllib.request import urlretrieve
from urllib.request import URLError
from urllib import request
import urllib
import json
import re

def extract(text, sub1, sub2):
    return text.split(sub1, 1)[-1].split(sub2, 1)[0]

word = input("Input key word: ")
word=urllib.parse.quote(word)
url = ''.join(["https://images.baidu.com/search/flip?tn=baiduimage&ie=utf-8&word=",
              word,"&ct=201326592&ic=0&lm=-1&width=&height=&v=flip"])

headers = {'User-Agent': 'User-Agent:Mozilla/5.0'}
req = request.Request(url, headers=headers)
data = urlopen(req)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    charset=data.info().get_content_charset()
    html = data.read()

    encoding = extract(str(html).lower(), 'charset=', '"')
    print('-'*50)
    print( "Encoding type = %s" % encoding )
    print('-'*50)

    html = html.decode(encoding)

    #data.close()

    #reg = r'src="(.+?\.jpg)"'
    imglist = re.findall('"objURL":"(.*?)",', html, re.S)

    for x, imgurl in enumerate(imglist):
        print(imgurl)
        try:
            urlretrieve(imgurl, "%s.jpg" % x)
        except URLError as e:
            if e.code == 404:
                print("4 0 4")
            else:
                print("%s" % e)
                continue
```

运行程序，输入关键字“动漫”后将开始爬取相应的图片，并保存到程序运行目录下：

```
Input key word: 动漫
----------------------------------------------------
Encoding type = utf-8
----------------------------------------------------
http://pic1.16pic.com/00/49/83/16pic_4983642_b.jpg
http://pic2.16pic.com/00/32/58/16pic_3258676_b.jpg
http://imgsrc.baidu.com/imgad/pic/item/38dbb6fd5266d016e40f4dfc9d2bd40734fa
35d4.jpg
http://imgsrc.baidu.com/imgad/pic/item/5d6034a85edf8db1621da2b00323dd54564e
7437.jpg
http://pic3.16pic.com/00/04/57/16pic_457847_b.jpg
http://pic2.16pic.com/00/04/39/16pic_439069_b.jpg
http://pic2.16pic.com/00/32/64/16pic_3264151_b.jpg
http://pic17.nipic.com/20111108/8687258_133750475000_2.jpg
…
```

14.2 email

14.2.1 smtplib 模块

email 是一个古老的但仍广泛使用的数字通信协议，Python 的标准库中包含了发送、接收和存储 email 消息的模块，其中，smtplib 模块负责和 SMTP 邮件服务器通信，用于发送邮件。SMTP（Simple Mail Transfer Protocol）是一个发送邮件和在邮件服务器转发邮件的协议。

用 smtplib 模块向 SMTP 服务器发送一个邮件主要分为三步：创建一个 SMTP 对象；认证登录 SMTP 服务器；使用 sendmail()方法发送邮件。

1．创建一个 SMTP 对象

创建 SMTP 对象的语法格式为：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import smtplib
smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

其中，参数 host 是运行 SMTP 服务器的主机名，既可以是 IP 地址，也可以是域名，如 smtp.gmail.com。可选参数 port 是邮件服务器的监听端口，默认是 25。如果 SMTP 服务器在发送邮件的本地机器上运行，则可以将 local_hostname 设置为 localhost。

2．认证登录 SMTP 服务器

通常，SMTP 服务器都需要登录验证，并且可能会对通信消息进行加密。因此，需要首先发送握手信号(通过 SMTP 对象调用 SMTP 的 echo()方法)，根据服务器返回的信息，再决定是否要登录服务器，以及是否要用 STARTTLS 协议加密消息。例如：

```
smtp.ehlo()
if smtp.has_extn('STARTTLS'): #如果需要 STARTTLS 加密
   #需要 SMTP 对象启动 STARTTLS 协议,并再次通过 echo()方法告知 SMTP 服务器
   smtp.starttls()
   smtp.ehlo()
else:
   pass  #什么也不需要做
if smtp.has_extn('AUTH'):
   #需要认证登录
   smtp.login(sender_email, password)
else:
   #不需要认证登录
```


如果根据 echo()方法返回的信息需要验证登录，则需要通过创建的 SMTP 对象调用 SMTP 的 login()方法登录 SMTP 服务器，login()方法规范如下：

```
smtp.login(sender_email, password)
```

其中，参数 sender_email 是发送者的 email 账号名，参数 password 是 email 密码或验证码(有的 SMTP 需要验证码代替密码登录 SMTP 服务器)。

3．使用 sendmail()方法发送邮件

sendmail()方法的语法格式为：

```
SMTP.sendmail(from_addr, to_addrs, msg, mail_options=(), rcpt_options=())
```

这个方法的三个必选参数是：参数 from_addr 是发送者邮箱；参数 to_addrs 是接收者邮箱；参数 msg 是邮件的内容(消息)。

下面是一个简单的发送 email 的程序(实际使用时请换成真正的服务器和邮箱名)：

```
import smtplib
sender = 'from@somedomain.com'          #发送者邮箱
receivers = ['to@somedomain.com']       #接收者邮箱
#邮件消息 message：包含邮件头和邮件内容
message = """From: From Person <from@somedomain.com>
To: To Person <to@somedomain.com>
Subject: SMTP email 标题

这里填写 email 的具体内容.
"""

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        print("成功发送了 email")
    except Exception:
        print("错误：不能发送 email")
```

4．数据加密

SSL 和 TLS 都是对通信双方之间的通信通道进行加密的标准协议。TLS 是 SSL 的后继者，通常这两个术语互换使用，除非需要指定特定的协议版本。STARTTLS 是将一个不安全的未加密的通信通道转换成一个加密的 SSL/TLS 通道的协议，STARTTLS 中使用的加密协议既可以是 TLS 也可以是 SSL。smtplib.SMTP_SSL 创建的 SMTP 对象建立的是一个采用 SSL/TLS 的加密的通信通道，smtplib.SMTP 创建的 SMTP 对象建立的是一个未采用 SSL/TLS 的未加密的通信通道，尽管该通道是未加密且不安全的通道，但是仍然可以通过 SMTP 对象调用 SMTP 的 starttls()方法将其包裹成一个采用 SSL/TLS 的加密消息的安全通道。

下面是一段简单的发送 SMTP 邮件消息的 Python 代码：

```
import smtplib
from email.mime.text import MIMEText

#1. 创建一个 SMTP 对象
USE_SSL = True              #SMTP_SSL 是加密的 SMTP
smtp_host = "smtp.zoho.com"
smtp_port = 587             #zoho 的 SMTP 端口，默认 SMTP 端口是 25，SMTP_SSL 端口是 465
smtp = None                 #SMTP 对象

if not USE_SSL:
    smtp_port =  587        #zoho 的 SMTP 服务器的端口
    smtp = smtplib.SMTP(smtp_host, smtp_port,timeout=30)
                            #连接 SMTP 邮件服务器,端口默认是 25
    #smtplib.SMTP('localhost')      #假设 SMTP 服务器就是发送者自己的机器
else:
    smtp_port = 465
    smtp = smtplib.SMTP_SSL(smtp_host, smtp_port,timeout=30)


#2. 检查 SMTP 服务器是否需要认证登录
smtp.ehlo()
if smtp.has_extn('STARTTLS'):
       print('(starting TLS)')
       smtp.starttls()
       smtp.ehlo()
else:
       print('(no STARTTLS)')

#3.登录
#定义发送的消息：发送者、接收者、密码或登录码
sender = 'xuepro@zoho.com'              #发送者
pwd =  'password'                       #pwd = '密码或登录码'
receivers = 'receriver@gmail.com'       #接收者列表

if smtp.has_extn('AUTH'):
    print('(logging in)')
    smtp.login(sender, pwd)
else:
    print('(无须认证登录)')
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
#4. 用 sendmail()方法发送消息
#消息内容：使用 MIMEText 构造符合 SMTP 协议的 header 及 body 的消息
content = 'hi,how are you ?'
msg = MIMEText(content,_subtype='plain',_charset='utf-8')
msg["Subject"] = "Hello"
msg["From"]    = sender
msg["To"]      = receivers

smtp.sendmail(sender, receivers, msg.as_string())   #发送邮件
smtp.quit()
```

输出：

```
(no STARTTLS)
(logging in)
(221, b'mx.zohomail.com closing connection')
```

注意

许多 STMP 邮箱需要验证码进行登录，密码处应该修改成“授权码”。

14.2.2 收取和处理邮件

收取邮件有两个协议，POP3(Post Office Protocol-Version 3，邮局协议版本 3)和 IMAP(Internet Message Access Protocol，互联网消息获取协议)，这两个协议都可以收取邮件。

POP3 允许在电子邮件客户端下载服务器上的邮件，但是在客户端的操作(如移动、删除邮件、标记已读等)不会反馈到服务器上。例如，通过客户端删除已读取的邮件，邮箱服务器上对应的邮件并没有被删除。

而 IMAP 提供服务器与邮件客户端之间双向通信时，客户端的操作都会反馈到服务器上，对邮件进行的操作，服务器上的邮件也会做相应的动作。IMAP 提供的摘要浏览功能可以让客户在阅读完所有邮件的到达时间、主题、发件人、大小等信息后再做出是否下载的决定。此外，IMAP 更好地支持了从多个不同设备中随时访问新邮件。IMAP 整体上为用户带来更为便捷和可靠的体验，而 POP3 更易丢失邮件或多次下载相同的邮件。例如，服务器对邮件只保留一定时间，超过时间就会自动删除邮件。

IMAP 除支持 POP 所有功能外，还支持以下功能：多个邮件文件夹(收件箱、发件箱、垃圾邮件……)；在 IMAP 服务器上进行标记，如 Seen、Replied、Read、Deleted；在服务器的文件夹之间复制和移动邮件等。

imaplib 模块提供了与 IMAP 服务器通信的客户的功能实现。IMAP 定义了一系列发送到服务器的命令和传回客户的响应。大多数与服务器通信的命令都是通过 IMAP4 类对象的方法提供的。关于 IMAP 协议可参考 RFC 3501 标准。

有三个类采用不同通信机制与服务器通信：IMAP4 类，使用明文的 socket 与服务器通信；IMAP4SSL 类，使用 SSL 套接字上的加密通信；IMAP4stream 类，使用外部命令的标准输入和标准输出。下面的示例使用 IMAP4_SSL，但其他类的 API 使用方法是类似的。

1. 概念说明

存储在 IMAP 邮件服务器上的邮件也称消息(Message)，如同资源管理器一样，这些邮件(消息)被组织成不同的文件夹(子文件夹)，这些文件夹习惯地被称为 mailbox(邮箱)。在下面的叙述中，邮件和消息(Message)是同一个概念，邮箱、文件夹、目录也是同一个概念。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

2．连接服务器

以下为连接服务器的代码：

```
import imaplib

hostname = 'imap.zoho.com'
#hostname = 'imap.qq.com'  #imap.zoho.com'
connection = imaplib.IMAP4_SSL(host=hostname) #,port=imaplib.IMAP4_SSL_PORT)
#登录账号
username = 'xuepro@zoho.com'
password = 'apassword'
try:
    connection.login(username, password)
except Exception as err:
    print('ERROR:', err)
```

注意

在使用 IMAP 或 POP3 前，必须登录服务器，并开启 IMAP 或 POP3 功能。国内许多邮箱还需要手机验证码。

3．邮件文件夹列表

通过 IMAP4_SSL 的 list()的方法可以获得邮箱服务器中 mailbox(邮箱)列表：

```
import pprint
typ, data = connection.list()
print('Response code:', typ)
print('Response:')
pprint.pprint(data)
```

输出：

```
Response code: OK
Response:
[b'(\\HasChildren) "/" "INBOX"',
 b'(\\Noinferiors \\Drafts) "/" "Drafts"',
 b'(\\Noinferiors) "/" "Templates"',
 b'(\\Noinferiors \\Sent) "/" "Sent"',
 b'(\\Noinferiors \\Junk) "/" "Spam"',
 b'(\\HasNoChildren \\Trash) "/" "Trash"',
 b'(\\HasNoChildren) "/" "Notification"',
 b'(\\HasNoChildren) "/" "NewsLetter"',
 b'(\\HasNoChildren) "/" "INBOX/Friends"',
 b'(\\HasNoChildren) "/" "INBOX/Business"']
```

返回的 list 中的每个元素表示一个 mailbox(邮箱)文件夹，元素包含标志、分隔符和名字三个部分。例如，(\HasNoChildren \Trash) "/" "Trash"第一部分是一个元组，说明这个 mailbox(邮箱)文件夹没有子文件夹(\HasNoChildren)，且这个 mailbox 是 Trash(垃圾)(\Trash)；第二部分是一个简单的分隔符/；第三部分是这个 mailbox 的名字(Trash)。

可以编写一个函数解析每个 list 元素的这三个部分：

```
import re
#正则表达式的?P 给 3 个分组分别一个名字：flags、delimiter、name
list_response_pattern = re.compile(
    r'\((?P<flags>.*?)\) "(?P<delimiter>.*)" (?P<name>.*)'
)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
def parse_list_response(line):
    match = list_response_pattern.match(line.decode('utf-8'))
    flags, delimiter, mailbox_name = match.groups()
    mailbox_name = mailbox_name.strip('"')
    return (flags, delimiter, mailbox_name)

for line in data:
    print('Server response:', line)
    flags, delimiter, mailbox_name = parse_list_response(line)
    print('Parsed response:', (flags, delimiter, mailbox_name))
```

输出：

```
Server response: b'(\\HasChildren) "/" "INBOX"'
Parsed response: ('\\HasChildren', '/', 'INBOX')
Server response: b'(\\Noinferiors \\Drafts) "/" "Drafts"'
Parsed response: ('\\Noinferiors \\Drafts', '/', 'Drafts')
Server response: b'(\\Noinferiors) "/" "Templates"'
Parsed response: ('\\Noinferiors', '/', 'Templates')
Server response: b'(\\Noinferiors \\Sent) "/" "Sent"'
Parsed response: ('\\Noinferiors \\Sent', '/', 'Sent')
Server response: b'(\\Noinferiors \\Junk) "/" "Spam"'
Parsed response: ('\\Noinferiors \\Junk', '/', 'Spam')
Server response: b'(\\HasNoChildren \\Trash) "/" "Trash"'
Parsed response: ('\\HasNoChildren \\Trash', '/', 'Trash')
Server response: b'(\\HasNoChildren) "/" "Notification"'
Parsed response: ('\\HasNoChildren', '/', 'Notification')
Server response: b'(\\HasNoChildren) "/" "NewsLetter"'
Parsed response: ('\\HasNoChildren', '/', 'NewsLetter')
Server response: b'(\\HasNoChildren) "/" "INBOX/Friends"'
Parsed response: ('\\HasNoChildren', '/', 'INBOX/Friends')
Server response: b'(\\HasNoChildren) "/" "INBOX/Business"'
Parsed response: ('\\HasNoChildren', '/', 'INBOX/Business')
```


list()方法的规范是：

```
IMAP4.list(self, directory='""', pattern='*')
```

其中，该方法列举目录(directory)中匹配模式串 pattern 的所有 mailbox(邮箱)。Directory 的默认值是顶层的根目录，pattern 默认匹配任何值。

例如，下面的代码列出 INBOX 这个 mailbox(邮箱)文件夹下的所有子文件夹，即 Friends 子文件夹("INBOX/Friends")。

```
import pprint
typ, data = connection.list(directory='INBOX')
print('Response code:', typ)
for line in data:
    print(line)
Response code: OK
b'(\\HasNoChildren) "/" "INBOX/Friends"'
b'(\\HasNoChildren) "/" "INBOX/Business"'
```

也可以设置 pattern 参数，用于在 directory 中查找匹配的子 mailbox(邮箱)文件夹：

```
import pprint
typ, data = connection.list(directory='INBOX',pattern='*ri*')
print('Response code:', typ)
for line in data:
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    print(line)
```

输出：

```
Response code: OK
b'(\\HasNoChildren) "/" "INBOX/Friends"'
```

如果不指定目录时则在根目录下查找匹配的文件夹(邮箱)：

```
import pprint
typ, data = connection.list(pattern='*Trash*')
print('Response code:', typ)
for line in data:
    print(line)
```

输出：

```
Response code: OK
b'(\\HasNoChildren \\Trash) "/" "Trash"'
```

4．邮箱文件夹状态

可以用 status()方法查询一个邮箱文件夹状态。如表 14-1 所示是其中的一些邮箱状态条件及其含义。

表 14-1　邮箱状态条件的含义

条　件	含　义
MESSAGES	邮箱文件夹中消息数目
RECENT	具有\Recent(最近)标志的消息数目
UIDNEXT	邮箱的下一个唯一标识符值
UIDVALIDITY	邮箱的唯一标识符有效性值
UNSEEN	不具有\Seen(已阅)标志的消息数目

可以将这些状态条件用空格隔开，构成一个字符串并传给 status()方法：

```
for line in data:
      flags, delimiter, mailbox = parse_list_response(line)
      print('Mailbox:', mailbox)
      status = connection.status(
          '"{}"'.format(mailbox),
          '(MESSAGES RECENT UIDNEXT UIDVALIDITY UNSEEN)',
      )
      print(status)
```

输出：

```
Mailbox: Trash
('OK', [b'"Trash" (MESSAGES 1 RECENT 1 UNSEEN 1 UIDNEXT 6 UIDVALIDITY 1)'])
```

以上程序返回的是一个 tuple 对象，第一个值是响应代码，然后是一个服务器返回的信息的 list。其中的一个元素是邮箱文件夹(mailbox)名字，还有一个 tuple 对象，这个 tuple 对象是一些状态条件及值。

5．选择一个邮箱

对邮件的基本操作通常是选择一个邮箱(如收件箱)，处理其中的邮件。IMAP 连接是一个有状态的连接，当选择某个邮箱后，后续对邮件的操作针对的一直是这个邮箱，直到选择了另外一个邮箱。例如：

```
typ, data = connection.select('Trash')
print(typ, data)
num_msgs = int(data[0])
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print('There are {} messages in Trash'.format(num_msgs))
```

输出：

```
OK [b'1']
There are 1 messages in Trash
```

响应信息中给出了这个邮箱中邮件的数目。

6．搜索邮件

选择了邮箱后，就可以通过 search()方法搜索邮件，语法格式如下：

```
IMAP4.search(charset, criterion[, ...])
```

第一个参数指定该请求的 charset(字符集)，默认是 None，表示不指定字符集，后面至少跟一个 criterion 搜索条件。例如，“ALL”表示所有的消息，“(SUBJECT "he")”表示在主题中搜索"he"。详细信息请参考 RFC 3501 标准。

该方法返回的是搜索的结果状态和搜索到的邮件的 id 的集合。例如：

```
typ, msg_ids = connection.search(None, 'ALL')
print('Trash', typ, msg_ids)
```

输出：


```
Trash OK [b'1']
```

7．获取邮件内容

可以用 fetch()方法获取一个邮件的组成部分(parts)，语法格式如下：

```
IMAP4.fetch(message_set, message_parts)
```

其中，参数 message_parts 是用圆括号括起来的，由消息的组成部分构成的字符串。例如，(UID BODY[TEXT])表示要获取消息的 UID，BODY[TEXT]表示获取消息体(BODY)中的正文部分，BODY.PEEK[HEADER]表示获取消息头。参数 message_set 是要获取的邮件消息的 list 列表，如"1,2"表示获取第一和第二个邮件消息。例如：

```
typ, msg_data = connection.fetch('1', '(BODY.PEEK[HEADER] FLAGS)')
#print(msg_data)
pprint.pprint(msg_data)
```

输出：

```
[(b'1 (FLAGS (\\Recent) BODY[HEADER] {107}',
  b'Subject: new message\r\nFrom: xuepro@zolo.com\r\nTo: xuea.to@gmail.com'
  b'\nX-ZohoMail-Sender: xuepro@zolo.com\r\n\r\n'),
 b')']
```

8．获取邮件的整个内容

上述讲解的是获取邮件消息的不同部分的内容，也可以获取 RFC 822 邮件消息格式的完整消息，并用 email 模块中的类 email.parser 解析它。例如：

```
import email
import email.parser

typ, msg_data = connection.fetch('1', '(RFC822)')
for response_part in msg_data:
       if isinstance(response_part, tuple):
          email_parser = email.parser.BytesFeedParser()
          email_parser.feed(response_part[1])
          msg = email_parser.close()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
                for header in ['subject', 'to', 'from']:
                    print('{:^8}: {}'.format(
                        header.upper(), msg[header]))
```

输出：

```
SUBJECT : new message
   TO   : xuea.to@gmail.com
  FROM  : xuepro@zolo.com
```

9. 添加消息

向邮箱中添加一个消息的方法是，构造一个消息对象并调用 IMAP4.append()方法，将该消息及消息的时间戳加到邮箱中。例如：

```
import email.message
import time
#构造一个消息对象
new_message = email.message.Message()
new_message.set_unixfrom('xuepro')
new_message['Subject'] = 'new message'
new_message['From'] = username@gmail.com'
new_message['To'] = 'xuepro@zolo.com'
new_message.set_payload('This is the body of the message.\n')
print(new_message)
#用 append()将消息及时间戳添加到邮箱'INBOX'中
connection.append('INBOX', '',
           imaplib.Time2Internaldate(time.time()),
           str(new_message).encode('utf-8'))
#选择邮箱'INBOX'
connection.select('INBOX')
typ, [msg_ids] = connection.search(None, 'ALL')         #搜索邮箱
for num in msg_ids.split():
       typ, msg_data = connection.fetch(num, '(BODY.PEEK[HEADER])') #获取邮件消息内容
       for response_part in msg_data:
           if isinstance(response_part, tuple):
               print('\n{}:'.format(num))
               print(response_part[1])
```

输出：

```
Subject: new message
From: username@gmail.com
To: xuepro@zolo.com

This is the body of the message.

b'1':
b'Subject: new message\r\nFrom: username@gmail.com\r\nTo: xuepro@zolo.com\nX-
ZohoMail-Sender: username@gmail.com\r\n\r\n'

b'2':
b'Subject: new message\r\nFrom: username@gmail.com\r\nTo: xuepro@zolo.com\nX-
ZohoMail-Sender: username@gmail.com\r\n\r\n'
```

10. 移动和复制消息

一旦消息已到达服务器，就可以用 move()方法或 copy()方法在不同邮件文件夹中移动或复制它们。其参数和 fetch()方法都是一组 id。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
connection.select('INBOX')
typ, [response] = connection.search(None, 'SEEN')
if typ != 'OK':
     raise RuntimeError(response)
msg_ids = ','.join(response.decode('utf-8').split(' '))

#创建一个名为 newMailBox 的新邮箱(mailbox)
new_mailbox = "newMailBox"
typ, create_response = connection.create(new_mailbox)
print('CREATED Example.Today:', create_response)

 #将 msg_ids 的消息复制到新邮箱 new_mailbox
print('COPYING:', msg_ids)
connection.copy(msg_ids, new_mailbox)

#选择新邮箱并显示其中所有邮件消息
connection.select(new_mailbox)
typ, [response] = connection.search(None, 'ALL')
print('COPIED:', response)
```

输出：

```
CREATED Example.Today: [b'[ALREADYEXISTS] create failure: Folder: /newMailBox exists']
COPYING: 1
COPIED: b'1 2'
```


11．删除消息(邮件)

通常，通过设置邮件的删除标志表示该邮件被删除，被删除的邮件只是做了一个标记，可以用 store() 方法修改一个邮件的删除标记，但并没有真正删除该邮件。如果要真正删除邮件，则需要调用 expunge() 方法。例如：

```
#查询邮箱中的所有邮件的 id
typ, [msg_ids] = connection.search(None, 'ALL')
print('Starting messages:', msg_ids)

#查询邮箱中的满足条件的邮件 id
typ, [msg_ids] = connection.search(
      None, '(SUBJECT "message")',
)
msg_ids = ','.join(msg_ids.decode('utf-8').split(' '))
print('Matching messages:', msg_ids)

#这些邮件消息的当前标志 FLAGS 是什么
typ, response = connection.fetch(msg_ids,'(BODY.PEEK[HEADER] FLAGS)')  #'(BODY.
PEEK[HEADER] FLAGS)'
print('Flags before:', response)

#给这些邮件消息设置删除标志
typ, response = connection.store(msg_ids, '+FLAGS', r'(\Deleted)')

#这些邮件消息的当前标志 FLAGS 是什么
typ, response = connection.fetch(msg_ids, '(FLAGS)')
print('Flags after:', response)

#真正删除这些邮件消息
typ, response = connection.expunge()
print('Expunged:', response)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
#邮箱中剩余哪些邮件消息
typ, [msg_ids] = connection.search(None, 'ALL')
print('Remaining messages:', msg_ids)
```

输出：

```
Starting messages: b'1 2'
Matching messages: 1,2
Flags before: [(b'1 (FLAGS (\\Recent \\Seen) BODY[HEADER] {111}', b'Subject: new
message\r\nFrom: username@gmail.com\r\nTo: xuepro@zolo.com\nX-ZohoMail-Sender:
username@gmail.com\r\n\r\n'), b')', (b'2 (FLAGS (\\Recent \\Seen) BODY[HEADER]
{111}', b'Subject: new message\r\nFrom: username@gmail.com\r\nTo: xuepro@zolo.com\
nX-ZohoMail-Sender: username@gmail.com\r\n\r\n'), b')']
Flags after: [b'1 (FLAGS (\\Deleted \\Recent \\Seen))', b'2 (FLAGS (\\Deleted
\\Recent \\Seen))']
Expunged: [b'1', b'1']
Remaining messages: b''
```

可以看到，在设置(store()方法)删除标志(Deleted)后，这些邮件仍然是存在的，直到执行了expunge()方法后，这些消息才从邮箱中真正地被删除了。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

第 15 章　数据持久化

数据持久性是指将内存中的不同数据类型的数据保存到外部存储器(如硬盘)上，并能够在需要时从外部存储器将这些数据恢复为内存中的数据类型，这个过程称为“**数据持久化**(**Data Persistence**)”。

保留数据以供长期使用涉及两个方面：

- 数据在内存中的对象和存储格式之间相互转换；
- 对不同格式的数据进行处理和存储。

前者称为“**序列化**(**serializing**)”，即将 Python 中不同类型的对象转换为字节流，并从字节流中恢复出 Python 对应类型的对象。标准库包括各种模块，可以处理以上两个方面的问题。例如，pickle 模块和 marshal 模块支持许多 Python 数据类型对象的序列化。shelve 模块可以将 pickle 模块序列化的字节流以字典(也称散列)的形式存储到外部存储器上，即可以通过 key 模块访问已序列化的对象。

json 模块是适合于网络数据传输的一种常见格式，Python 的 json 模块支持对象和 json 文件之间的持久化。

另外，普通的文件和各种数据库也支持永久保存数据，如 dbm 和 sqlite 等数据库。

15.1　pickle 模块

pickle 模块实现用于序列化和反序列化 Python 对象结构的二进制协议。“pickling”是将 Python 对象层次结构转换为字节流的过程，“unpickling”是反向操作，即将字节流(来自二进制文件或类似字节的对象)转换回对象层次结构。“pickling”也称为“序列化”“编组”或“扁平化”，而“unpickling”称为“反序列化”。为避免混淆，应尽量使用术语“pickling”和“unpickling”。

pickle 模块有两类函数 dump()和 load()分别执行“pickling” 和 “unpickling”操作。

pickle.dumps(obj, protocol=None, *, fix_imports=True)将一个 Python 对象转换成一个 bytes 类型的字节串对象。

pickle.dump(obj, file, protocol=None, *, fix_imports=True)将一个 Python 对象转换成一个 bytes 类型字节串对象，并保存到文件中。

pickle.loads(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict")将已经序列化(picked)的 bytes 类型字节串 bytes_object 恢复成 Python 类型对象。

pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict")从打开的文件对象中读取已经序列化(picked)的 bytes 字节串，并恢复为 Python 类型对象。

其中的参数说明如下。

- obj：要持久化保存的对象。
- file：可以是一个以读/写模式打开的文件对象、一个 io.BytesIO 对象，或者其他自定义的满足条件的对象。对于函数 dump()，file 参数对象必须有一个 write()方法，该方法接收一个字节串作为参数。对于函数 load()，file 参数对象必须有一个 read()方法和 readline()方法，用于读取一定数量的字符或一行字符，read()方法和 readline()方法都返回读取的字节流。
- bytes_object：表示一个字节对象。
- protocol：这是一个可选的参数，默认为 0，如果设置为 1 或 True，则以高压缩的二进制格

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

式保存持久化后的对象，否则以 ASCII 格式保存。

- fix_imports：用于处理 Python3 和 Python2 的名字兼容性问题，可暂不考虑。
- encoding：默认是“ASCII”，也可以是“‘bytes”（字节）。

pickle 模块还提供两个专门的类，Pickler 和 Unpickler。这两个类分别有两个方法 dump() 和 load() 用于执行和上述函数类似的序列化（pickling）和反序列化（unpickling）操作。

下面看一个具体例子：

```
import pickle

obj = {"书名":'C++17 从入门到实践' , "价格": 39.0, "作者": 'hwdong'}
obj2 = (1,2,3,4)

f = open('obj.pkl', 'wb')
pickle.dump(obj, f)
pickle.dump(obj2, f)
pickle.dump("hello world", f)
f.close()

f = open('obj.pkl', 'rb')
x1 = pickle.load(f)
x2 = pickle.load(f)
x3 = pickle.load(f)
f.close()
print(x1)
print(x2)
print(x3)
```

输出：

```
{'书名': 'C++17 从入门到实践', '价格': 39.0, '作者': 'hwdong'}
(1, 2, 3, 4)
hello world
```

也可以序列化对象到一个字节串对象中，这个序列化的字节串对象随后可以写到文件、网络、管道等。例如：

```
import pickle
obj = {"书名":'C++17 从入门到实践' , "价格": 39.0, "作者": 'hwdong'}
print(obj)

byte_object = pickle.dumps(obj)
print('pickling: \n{!r}'.format(byte_object))
```

输出：

```
{'书名': 'C++17 从入门到实践', '价格': 39.0, '作者': 'hwdong'}
pickling:
b'\x80\x03}q\x00(X\x06\x00\x00\x00\xe4\xb9\xa6\xe5\x90\x8dq\x01X\x12\x00\x00
\x00python\xe7\xa8\x8b\xe5\xba\x8f\xe8\xae\xbe\xe8\xae\xa1q\x02X\x06\x00\x00
\x00\xe4\xbb\xb7\xe6\xa0\xbcq\x03G@C\x80\x00\x00\x00\x00\x00X\x06\x00\x00\x00
\xe4\xbd\x9c\xe8\x80\x85q\x04X\x06\x00\x00\x00hwdongq\x05u.'
```

用函数 load() 同样可以反序列化恢复出原来的对象。例如：

```
import pickle
obj = {"书名":'C++17 从入门到实践' , "价格": 39.0, "作者": 'hwdong'}
print(obj)
bytes_object = pickle.dumps(obj)
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
x = pickle.loads(bytes_object)

print('恢复的结果: ')
print(x)
```

输出：

```
{'书名': 'C++17 从入门到实践', '价格': 39.0, '作者': 'hwdong'}
恢复的结果:
{'书名': 'C++17 从入门到实践', '价格': 39.0, '作者': 'hwdong'}
```

可以将一个对象序列化到一个与文件类似的**流对象**中，也可从一个流对象反序列化重构原来的对象。例如，下面的程序将学生数据序列化到一个 io.BytesIO 字节流对象，再从另外一个字节流对象中反序列化出学生数据，当然，这个序列化的字节流对象可以存储到文件，或者发送到网络：

```
import io
import pickle
class Student:
    def __init__(self, name,score):
        self.name = name
        self.score = score
        return
data = []
data.append(Student('Wang',30.5))
data.append(Student('Li',60.5))
data.append(Student('Zhao',90.5))

#模拟一个文件流
out_s = io.BytesIO()

#序列化数据到一个流对象
for o in data:
    pickle.dump(o, out_s)
    out_s.flush()

#新建一个新的字节流对象
in_s = io.BytesIO(out_s.getvalue())

#从一个流对象反序列化数据
while True:
    try:
        o = pickle.load(in_s)
    except EOFError:
        break
    else:
        print('读取    : {} ({})'.format(
            o.name, o.score))
```

输出：

```
读取    : Wang (30.5)
读取    : Li (60.5)
读取    : Zhao (90.5)
```

最后需要说明的是，不是所有的对象都可以序列化，套接字、文件句柄、数据库连接，以及依赖操作系统或其他进程的其他对象等都不能用其原始数据序列化。具有不可序列化属性的对象可以首先通过定义两个函数，__getstate__() 和 __setstate__()，将这些状态转化为可以序列化的状态(属性)，然后再序列化这些对象。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

15.2 shelve 模块

shelve 模块以字典(散列)的方式存储或读取序列化对象，即它以一个“key-value(键-值)”形式保存数据，其中的 value(值)是一个可以用 pickle 模块序列化的任何对象(类类型、递归数据类型)，其中的 key(键)可以是任意字符串。value 被 pickle 序列化和反序列化并被写入由 dbm 模块创建和管理的数据库，当不需要关系数据库时，shelve 模块可以用作 Python 对象的简单持久的存储选项。

通过 shelve 模块的函数 open()打开一个持久化的字典的格式如下：

```
shelve.open(filename, flag='c', protocol=None, writeback=False)
```

如果可选参数 writeback 设置为 True，则所有访问的条目会缓存在内存中，并可通过 sync()和 close()写回到文件中。

另外，如果参数 filename 没有文件扩展名，则函数 open()会创建后缀为.bck、.dat、.dir 的三个文件。

下面程序将两个对象(obj 和 obj2)分别用两个键“zhang”和“wang”保存到 shelve 对象 db 中：

```
import shelve

obj = {'name':'Zhang','score':90.5}
obj2 = ['王',60.5,30]
db = shelve.open('shelf.db')  #打开一个文件
db['zhang'] = obj   #给 key 为'zhang'的 key-value 一个值 obj
db['wang'] = obj2   #给 key 为'wang'的 key-value 一个值 obj2
db.close()   #关闭文件
```

重新打开上面文件并根据 key(键)读取内容到对象 obj 和 obj2 中：

```
import shelve

db = shelve.open('shelf.db')  #打开一个文件
obj = db['zhang']
obj2 = db['wang']
print(obj)
print(obj2)
db.close()   #关闭文件
```

输出：

```
{'name': 'Zhang', 'score': 90.5}
['王', 60.5, 30]
```

writeback 默认值是 False，即“不可回写”，也就是说，一个 key-value 中的 value 是不可以修改的，但是可以将该 key 的值替换成其他的值。例如：

```
import shelve
db = shelve.open('shelf.db')       #打开一个文件
#db 打开时  writeback=False
db['wang'].append(78)              #不可以修改 key-value 中的 value
print(db['wang'])
db['wang'] = [34,78,1,100]         #但该 key 的 value 可替换成其他的 value
print(db['wang'])
db.close()
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
['王', 60.5, 30]
[34, 78, 1, 100]
```

如果打开文件时，设置 writeback=True，那么就可以修改 key-value 中的 value：

```
import shelve
db = shelve.open('shelf.db',writeback=True)     #打开一个文件

#db 打开时  writeback=True
db['wang'].append(111)                          #可以修改 key-value 中的 value
print(db['wang'])
db.sync()     #如果 writeback=True,那么可以对内存 db 进行修改，高速缓存中的所有条目写回
              #文件中该方法会自动被 close()方法调用
```

输出：

```
[34, 78, 1, 100, 111]
```

删除一个不存在的键：

```
del db['zhang']            #删除一个键
print(db['zhang'])         #抛出 KeyError 异常
db.close()
```


将抛出 KeyError 类型的异常：

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-26-f0ad236d1819> in <module>()
----> 1 del db['zhang']        #删除一个键
      2 print(db['zhang'])     #抛出 KeyError 异常
      3 db.close()
...
```

可以用 with 语句，以防止忘记调用 sync()方法和 close()方法。例如：

```
import shelve
with shelve.open('shelf2.db') as s:
   s['zhang'] = {
       '名字': 'Zhang',
       '分数': 90.5,
       '年龄': 23,
   }
with shelve.open('shelf2.db') as s:
   print(s['zhang'])
```

输出：

```
{'名字': 'Zhang', '分数': 90.5, '年龄': 23}
```

15.3 dbm 模块

所有版本的 Linux 及大多数的 UNIX 版本都随系统自带一个以键-值(key-value)形式存储的 DBM 数据库。例如，在 ubuntu 系统上，可以通过下面命令安装：

```
sudo apt-get install libgdbm-dev
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Python 的 dbm 模块是 DBM 数据库(dbm.gnu 或 dbm.ndbm)的通用接口。如果没有安装这些 DBM 数据库，则将使用简单的但速度较慢的模块 dbm.dumb 作为 DBM 数据库。

和 shelve 模块不同的是，dbm 模块中的键(key)和值(value)都必须是字符串，而 shelve 模块中的 value 可以是任何 Python 类型对象。

dbm.gnu、dbm.ndbm 和 dbm.dumb 模块的函数 open()都具有一致的接口，其格式如下：

```
open(filename[, flag[, mode]])
```

其中，

可选的标志参数 flag 必须是以下值之一：

'r'：打开一个存在的文件(默认值)。

'w'：打开文件对其读/写，如果文件不存在，则不会创建它。

'c'：打开文件对其进行读/写，如果不存在则创建该文件。

'n'：总是创建一个新的空白文件用于读/写。

可选的参数 mode 是文件的 Unix 模式，仅在需要创建数据库时使用。它默认为八进制的 0o666。

下面程序创建一个新的空数据库，并写入一些“键–值”：

```
import dbm
with dbm.open('test_dbm.db', 'n') as db:
    db['name'] = '张伟'
    db['ege'] = '23'
    db['分数'] = '68.9'
```

打开这个存在的数据库，并通过 keys()方法遍历其中的 key(键)：

```
import dbm
with dbm.open('test_dbm.db', 'r') as db:
    print('keys():', db.keys())

    for k in db.keys():
        print('iterating:', k, db[k])

    print('db["分数"] =', db['分数'])
```

输出：

```
keys(): [b'name', b'ege', b'\xe5\x88\x86\xe6\x95\xb0']
iterating: b'name' b'\xe5\xbc\xa0\xe4\xbc\x9f'
iterating: b'ege' b'23'
iterating: b'\xe5\x88\x86\xe6\x95\xb0' b'68.9'
db["分数"] = b'68.9'
```

15.4 json 模块

JSON(JavaScript Object Notation)是一种轻量级的数据交换格式，既易于人阅读和编写，也易于机器解析和生成。JSON 采用完全独立于语言的文本格式，这些特性使 JSON 成为 Web 等应用的理想的数据交换语言。JSON 和 Pickle 有以下区别。

- JSON 是以文本形式存储的，Pickle 则是以二进制形式存储的。
- JSON 是人可读的，Pickle 是人不可读的。
- JSON 广泛应用于除 Python 外的其他领域，Pickle 是 Python 独有的。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

- JSON 只能序列化一些 Python 的内置对象，Pickle 几乎可以存储所有对象。

Python 的 json 模块能够理解 Python 的数据类型(如 str、int、float、list、tuple 和 dict 等)，支持 Python 对象和 json 字符串格式之间的相互转换。

Json 模块的 API 函数类似 pickle 模块，主要有序列化的函数 dumps()和反序列化的函数 loads()。

15.4.1 简单数据类型的编码和解码

1. 编码

可以用函数 json.dumps()将 Python 对象编码转换为字符串形式。例如：

```
import json
python_obj = [[1,2,3],3.14,'abc',{'key1':(1,2,3),'key2':[4,5,6]},True,False,None]
json_str=json.dumps(python_obj)
print(json_str)
```

输出：

```
[[1, 2, 3], 3.14, "abc", {"key1": [1, 2, 3], "key2":
[4, 5, 6]}, true, false, null]
```

表 15-1 Python 原始类型和 json 类型的转换

Python	json
dict	object
list、tuple	array
str、unicode	string
int、long、float	number
true	true
False	false
None	null

简单类型对象编码后的字符串和其原始的 repr()结果基本是一致的，但有些数据类型，如上例中的元组(1, 2, 3)被转换成了[1, 2, 3](json 模块的 array 数组形式)。Python 原始类型和 json 类型的转化，可以参考表 15-1。

可以向函数 json.dumps()传递一些参数以控制转换的结果。例如，参数 sort_keys=True 时，dict 类型的数据将按 key(键)有序转换：

```
data = [{'xyz': 3.0,'abc': 'get', 'hi': (1,2) },'world','hello']
json_str = json.dumps(data)
print(json_str)
json_str = json.dumps(data, sort_keys=True)
print(json_str)
```

输出：

```
[{"xyz": 3.0, "abc": "get", "hi": [1, 2]}, "world", "hello"]
[{"abc": "get", "hi": [1, 2], "xyz": 3.0}, "world", "hello"]
```

即当 sort_keys=True 时，转换后的 json 串对于字典的元素是按键(key)有序的。

对于结构化数据，可以给参数 indent 设置一个值(如 indent=3)来产生具有缩进的、阅读性好的 json 串：

```
json_str = json.dumps(data, sort_keys=True,indent = 3)
print(json_str)
```

输出：

```
[
   {
      "abc": "get",
      "hi": [
         1,
         2
      ],
      "xyz": 3.0
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        },
        "world",
        "hello"
    ]
```

这种格式类似美观输出函数 pprint()的输出字符串。

2．dict 编码的问题

json 格式期望 dict(字典)的 key 应该是字符串类型，而对于 key 不是(或不能自动转化)字符串类型的 dict 对象，则不能直接进行编码，且会抛出 TypeError 错误。例如：

```
data = [{1: 2, 'hi':3,(4,): 'A'}]
json_str = json.dumps(data)
print(json_str)
```

抛出了异常“TypeError: keys must be a string”。

可以将函数 json.dumps()的参数 skipkeys 设置为 skipkeys=True，即跳过非字符串的 key(键)：

```
data = [{1: 2, 'hi':3,(4,): 'A'}]
json_str = json.dumps(data,skipkeys=True)
print(json_str)
```

跳过了键为(4,)的元素，而键为 1 的整数被转换为字符串"1"。

输出：

```
[{"1": 2, "hi": 3}]
```

3．解码

函数 json.loads()将 json 格式字符串解码转换为 Python 对象。例如：

```
import json
python_obj = [[1,2,3],3.14,'abc',{'key1':(1,2,3),'key2':[4,5,6]},True,False,None]
json_str=json.dumps(python_obj) #转换为 json 格式字符串
print(json_str)

python_obj_recovered =  json.loads(json_str) #将 json 格式字符串转换为 Python 对象
print(python_obj_recovered)
```

输出：

```
[[1, 2, 3], 3.14, "abc", {"key1": [1, 2, 3], "key2": [4, 5, 6]}, true, false, null]
[[1, 2, 3], 3.14, 'abc', {'key1': [1, 2, 3], 'key2': [4, 5, 6]}, True, False, None]
```

15.4.2 自定义类型的编码和解码

json 模块能自动识别 Python 内在类型，但无法直接处理用户定义类型的对象，对于用户定义类型对象，通常有两种方法进行编码和解码。

假设有如下的自定义类(保存在 myobj.py 模块中)：

```
#myobj.py
class Point:
    def __init__(self, x=0,y=0):
        self.x = x
        self.y = y
    def __repr__(self):
        return '<Point({},{})>'.format(self.x,self.y)
```

最简单的方法是定义一个函数，将自定义类型转换为已知的内在类型，并将这个函数作为 json 的编码函数 json.dumps()的参数：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import json
import point
obj = point.Point(3,4)
try:
    print(json.dumps(obj))
except TypeError as err:
    print('ERROR:', err)

def convert_to_dict(obj):
    #将这个对象转化为一个 dict 对象
    d = {
        '__class__': obj.__class__.__name__,
        '__module__': obj.__module__,
    }
    d.update(obj.__dict__)
    return d
print(json.dumps(obj, default=convert_to_dict))
```

一个对象的__dict__是用来存储对象属性的字典对象，其键为属性名，其值为属性的值。函数convert_to_dict()根据形参对象 obj，创建键为__class__和__module__的 dict 对象，接着用 obj 对象的__dict__属性更新 dict 对象。最后，将这个函数 convert_to_dict()作为 dumps()函数的 default 值。

程序的输出：

```
ERROR: Object of type 'Point' is not JSON serializable
{"__class__": "Point", "__module__": "point", "x": 3, "y": 4}
```

反过来，要从 json 字符串恢复创建一个对象，也应该相应地编写一个函数传递给解码函数 json.loads()：

```
import json
#从一个 jsong 字符串重建一个 Python 对象
def dict_to_object(d):
    if '__class__' in d:
        class_name = d.pop('__class__')
        module_name = d.pop('__module__')
        module = __import__(module_name)
        print('MODULE:', module.__name__)
        class_ = getattr(module, class_name)
        print('CLASS:', class_)
        args = {
            key: value
            for key, value in d.items()
        }
        print('INSTANCE ARGS:', args)
        inst = class_(**args)
    else:
        inst = d
    return inst

encoded_object = '''
    {"__class__": "Point", "__module__": "point", "x":3, "y":4}
    '''

recovered_obj = json.loads(
    encoded object,
    object_hook=dict_to_object,
)
print(recovered_obj)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

输出：

```
MODULE: point
CLASS: <class 'point.Point'>
INSTANCE ARGS: {'x': 3, 'y': 4}
<Point(3,4)>
```

函数 dict_to_object ()从 json 字符串中读取对象的类名、模块名、属性名及值，然后通过__import__(module_name)动态加载模块，并用函数 getattr(module, class_name)得到该模块中的类 class__，最后创建这个类 class__的对象(class__(**args))。

15.4.3 编码类和解码类

除编写专门的函数来转换自定义类对象外，Python 还定义了编码类 JSONEncoder 和 JSONDecoder 以分别用于编码和解码。例如：

```
import json
encoder = json.JSONEncoder()
data = [{'a': 'A', 'b': (2, 4)}]
print("原来的数据：",data)

json_data = json.JSONEncoder().encode(data)
print("编码的数据：",json_data)
x = json.JSONDecoder().decode(json_data)
print("解码的数据：",x)
```


输出：

```
原来的数据：[{'a': 'A', 'b': (2, 4)}]
编码的数据：[{"a": "A", "b": [2, 4]}]
解码的数据：[{'a': 'A', 'b': [2, 4]}]
```

JSONEncoder 还可以使用一个迭代的接口函数 iterencode()产生块形式的编码数据，使数据可以方便地通过网络或文件传输：

```
import json
encoder = json.JSONEncoder()
data = [{'a': 'A', 'b': (2, 4)}]

for part in encoder.iterencode(data):
   print(part,end='\t')
```

输出：

```
[   {   "a" :   "A" ,   "b" :   [2  , 4 ]   }   ]
```

如果要编码任意的对象，则可以定义一个 JSONEncoder 的派生类并重载实现 default()方法，default()方法类似上面的对自定义类编码的函数 convert_to_dict()：

```
import json
import point
class MyEncoder(json.JSONEncoder):
   def default(self, obj):
      d = {
         '__class__': obj.__class__.__name__,
         '__module__': obj.__module__,
      }
      d.update(obj.__dict__)
      return d

p = point.Point(3,4)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
print(p)
print(MyEncoder().encode(p))
```

输出：

```
<Point(3,4)>
{"__class__": "Point", "__module__": "point", "x": 3, "y": 4}
```

如果要解码一个自定义类对象的 json 字符串，则可以定义一个从 json.JSONDecoder 派生的从派生类，如 MyDecoder，并需要编写一个类似上例的函数 dict_to_object()的 dict_to_object()方法，另外还要在构造函数里设置以下解码方法：

```
import json
class MyDecoder(json.JSONDecoder):
    def __init__(self):
        json.JSONDecoder.__init__(self,
            object_hook=self.dict_to_object,
        )

    def dict_to_object(self, d):
        if '__class__' in d:
            class_name = d.pop('__class__')
            module_name = d.pop('__module__')
            module = __import__(module_name)
            print('MODULE:', module.__name__)
            class_ = getattr(module, class_name)
            print('CLASS:', class_)
            args = {
                key: value
                for key, value in d.items()
            }
            print('实例参数:', args)
            inst = class_(**args)
        else:
            inst = d
        return inst

encoded_object = '''
{"__class__": "Point", "__module__": "point", "x":3, "y":4}
'''

p2 = MyDecoder().decode(encoded_object)
print(p2)
```

输出：

```
MODULE: point
CLASS: <class 'point.Point'>
实例参数: {'x': 3, 'y': 4}
<Point(3,4)>
```

15.4.4 流或文件

可以将 Python 对象编码到流或文件，并从流或文件中解码重建 Python 对象，相应的函数是 dump()和 load()，其编码和解码过程类似编码和解码到一个字符串，同时必须传递一个流或文件对象作为参数。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

1．io.StringIO

```
import io
import json

data = [{'xyz': 3.0,2: 'get', 'hi': (1,2) },'world',True]

#构造一个输出流对象
f = io.StringIO()
json.dump(data, f)

print(f.getvalue())

#构造一个输入流对象
#f = io.StringIO('[{"a": "A", "c": 3.0, "b": [2, 4]}]')
f = io.StringIO(f.getvalue())
print(json.load(f))
```

输出：

```
[{"xyz": 3.0, "2": "get", "hi": [1, 2]}, "world", true]
[{'xyz': 3.0, '2': 'get', 'hi': [1, 2]}, 'world', True]
```

2．文件

同样，下面的代码将 Python 对象编码到文件中，并从文件中解码到另外一个对象中。

```
import json
obj = [{'xyz': 3.0,2: 'get', 'hi': (1,2) },'world',True]
f = open('data.json', 'w')
json.dump(obj, f)
f.close()

obj_recovered = json.load(open('data.json', 'r'))
print(obj_recovered)
```

输出：

```
[{'xyz': 3.0, '2': 'get', 'hi': [1, 2]}, 'world', True]
```

总结

- pickle 模块提供序列化和反序列化 Python 对象到一个文件和流对象的功能。
- shelve 模块的函数可以以 key-value 的形式存储或读取序列化对象，dbm 模块也是以 key-value 形式存储 DBM 数据库的条目，但 key 和 value 都必须是字符串，而 shelve 中的 value 可以是任何 Python 类型对象。
- JSON 是一种轻量级的数据交换格式，Python 的 json 模块支持 Python 对象和 json 字符串格式之间的相互转换。和 pickle 是二进制格式不同，json 是可读的文本格式。

15.5 sqlite3 模块

SQLite 是一个 C 库，它提供了一个轻量级的基于磁盘的数据库，它不需要单独的服务器进程，并允许使用 SQL 查询语言访问数据库。一些应用程序既可以使用 SQLite 进行内部数据存储，也可以使用 SQLite 对应用程序进行原型设计，并将代码移植到更大的数据库中，如 PostgreSQL 或 Oracle。

SQLite 是一个软件库，它实现了一个自包含的、无服务器的、零配置的事务性 SQL 数据库引擎。SQLite 是目前部署最广泛的 SQL 数据库引擎。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

SQLite 数据库作为单个文件存储在文件系统上，该数据库管理对文件的访问，管理方式包括锁定它以防止多个用户使用该文件时发生损坏。

Python 自带的 sqlite3 模块由 Gerhard Haring 编写，它提供了一个符合 PEP 249 描述的 DB-API 2.0 规范的 SQL 接口。

下列代码可检查 sqlite 的版本：

```
import sqlite3
sqlite3.sqlite_version
```

输出：

```
'3.21.0'
```

15.5.1 数据库基本操作

1. 创建数据库

通过 sqlite3 模块的函数 connect()传递一个代表数据库的文件名(company.db 文件)，并创建一个代表数据库的连接对象。例如：

```
import sqlite3
db_filename = 'company.db'
conn = sqlite3.connect(db_filename)
```


第一次尝试连接数据库时，会自动创建一个新的数据库文件(如代码中的 company.db 文件)。当然，对于一个新的数据库，它还没有用于存储数据的表(Table)。

2. 创建表

和所有关系数据库一样，sqlite 数据库由一些表组成，每个表就是类似学生表、工资单这样的，由一行行记录构造的表格。

一旦有了一个数据库的连接，就可以获得一个 Cursor 对象(游标对象)，通过调用 Cursor 对象的 execute()方法可执行 SQL 命令。游标对象可以产生一致的数据视图，这是与 SQLite 等事务数据库系统进行交互的主要手段。

首先，执行创建数据库表的 SQL 语句“CREATE TABLE...”创建一个表：

```
c = conn.cursor()    #通过连接 conn 的 cursor()函数获得一个"游标"
c.execute('''CREATE TABLE Department
          (name  text primary key, date text, description text)''')
```

创建名字叫作 Department 的表，这个表的数据项(列)是 name、date、description，它们的数据类型都是 text 文本，并且 name 是这个表的主键(primary key)。

下面的语句创建名字叫作 Employee 的表，这个表的数据项(列)是 id、name、date、salary 和 department。其中，salary 的类型是 real(实数)，department 项引用的是表 Department 的 name：

```
c.execute('''CREATE TABLE Employee
          (id  integer primary key autoincrement not null,
          name   text, date text, salary real,
          department text not null references Department(name))''')
```

执行上述代码后输出：

```
<sqlite3.Cursor at 0x21a00706500>
```

3. 插入行

创建表后可以执行插入行(rows)的 SQL 操作“INSERT INTO...”向表中添加数据记录。通过游标对象执行 Cursor 的 execute()方法，每次插入一行数据。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
#将两行数据插入 Department 表
c.execute("INSERT INTO Department VALUES ('Research','2006-01-05','产品研发')")
c.execute("INSERT INTO Department VALUES ('Sell','2008-01-05','sell productions')")

#将三行数据插入 Employee 表
c.execute("INSERT INTO Employee (name, date, salary, department)\
        VALUES ('Wang wei','2006-01-05',8000.5,'Research')")
c.execute("INSERT INTO Employee (name, date, salary, department)\
        VALUES ('Zhang ping','2008-01-05',7000.5, 'Sell')")
c.execute("INSERT INTO Employee (name, date, salary, department)\
        VALUES ('Zhao qiang ','2018-11-05',6000.5, 'Sell')")

#保存(提交 commit) 修改
conn.commit()

#如果完成了对数据库文件的操作，则必须通过 conn 的.close()方法关闭连接
#确保修改已经提交，否则数据会丢失
conn.close()
```

现在，数据库的两个表中都有相应的记录了，所以可以在命令行执行如下命令，用于从其中的一个表(如 Employee)提取出所有数据(所有行)。例如：

```
sqlite3 company.db "select * from Employee"
```


输出：

```
1|Wang wei|2006-01-05|8000.5|Research
2|Zhang ping|2008-01-05|7000.5|Sell
3|Zhao qiang |2018-11-05|6000.5|Sell
```

4．获取数据

如果要检索保存在数据库表中的数据，则同样要从创建的数据库连接得到 Cursor 对象(游标对象)，通过游标对象执行 Cursor 的 execute()方法来执行 SQL 选择(select)语句，告诉数据库引擎从哪个表里选择哪些列。再通过 Cursor 的 fetchall()方法来获得满足 select 子句的所有数据。fetchall()方法的返回值是一个元组序列，每个元组包含一条记录了由 select 子句指定的列的值。例如：

```
import sqlite3
db_filename = 'company.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
    select id, name, date, salary from Employee
    where department = 'Sell'
    """)

    for row in cursor.fetchall():
        Employee_id, name, date, salary = row
        print('{:2d} {:<25} ({}) [{:<8}] '.format(
            Employee_id, name, date, salary))
```

输出：

```
2 Zhang ping                (2008-01-05) [7000.5  ]
3 Zhao qiang                (2018-11-05) [6000.5  ]
```

也可以使用 fetchone()一次获得一条检索记录，或者使用 fetchmany()获得固定条目的记录。例如：

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
import sqlite3
db_filename = 'company.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
     select name, date, description from Department
     where name = 'Sell'
    """)

    name, date, description = cursor.fetchone()

    print('Department details for {} \n ({})  {}'.format(
        name, date, salary))

    cursor.execute("""
    select id, name, date, salary from Employee
    where department = 'Sell' order by date
    """)

    print('\nNext 5 employees:')
    for row in cursor.fetchmany(5):
        Employee_id, name, date, salary = row
        print('{:2d} {:<25} ({}) [{:<8}]'.format(
            Employee_id, name, date, salary))
```


输出：

```
Department details for Sell
 (2008-01-05)  6000.5

Next 5 employees:
 2 Zhang ping              (2008-01-05) [7000.5  ]
 3 Zhao qiang              (2018-11-05) [6000.5  ]
```

传递给 fetchmany()的值是要返回的最大记录数。如果表中没有足够的记录数，则返回的记录数将小于这个最大值。

Cursor 的 fetchone()、fetchmany()、fetchall()分别返回 Cursor 中的结果集中的一个、多个或所有记录。

5．更新数据

执行 SQL 的更新(update)子句可以修改表的记录。例如：

```
import sqlite3
db_filename = 'company.db'
with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()
    #更新 name= 'Wang wei'记录
    cursor.execute("""
     update Employee set date='2016-01-02'
     where name = 'Wang wei'
    """)

    conn.commit()   #提交修改到数据库中

    cursor.execute("""
    select id, name, date, salary from Employee
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    order by date
    """)

    print('\nNext 5 employees:')
    for row in cursor.fetchmany(5):
        Employee_id, name, date, salary = row
        print('{:2d} {:<25} ({}) [{:<8}]'.format(
            Employee_id, name, date, salary))
```

输出：

```
Next 5 employees:
 2 Zhang ping              (2008-01-05) [7000.5  ]
 1 Wang wei                (2016-01-02) [8000.5  ]
 3 Zhao qiang              (2018-11-05) [6000.5  ]
```

注意

执行完 SQL 操作后，只有通过连接对象提交(conn.commit())对数据库的修改，才真正生效。

6．删除

可以用 SQL 的 DELETE 子句删除表中的记录。例如：

```
import sqlite3
db_filename = 'company.db'
with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()
    cursor.execute("""
     DELETE from Employee
     where id = 3
    """)

    conn.commit()   #提交修改到数据库中

    cursor.execute("""
    select id, name, date, salary from Employee
    order by date
    """)

    print('\nNext 5 employees:')
    for row in cursor.fetchmany(5):
        Employee_id, name, date, salary = row
        print('{:2d} {:<25} ({}) [{:<8}]'.format(
            Employee_id, name, date, salary))
```

输出：

```
Next 5 employees:
2 Zhang ping                (2008-01-05) [7000.5  ]
1 Wang wei                  (2016-01-02) [8000.5  ]
```

7．查询元数据

根据 DB-API 2.0 规范，在调用 execute()方法之后，Cursor 会设置它的 description(描述属性)来保存由 fetch()方法返回的数据的信息。描述属性是包含“列名称、类型、显示大小、内部大小、精度、比例、是否接收空值的标志”的一系列元组。例如：

```
import sqlite3
db_filename = 'company.db'
with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    cursor.execute("""
    select * from Employee where department = 'Sell'
    """)

    print('Employee 表中包含这些列:')
    for colinfo in cursor.description:
        print(colinfo)
```

输出：

```
Employee 表中包含这些列:
('id', None, None, None, None, None, None)
('name', None, None, None, None, None, None)
('date', None, None, None, None, None, None)
('salary', None, None, None, None, None, None)
('department', None, None, None, None, None, None)
```

也可以用 SQL 命令“pragma table_info”获得一个表的信息：

```
import sqlite3
db_filename = 'company.db'
table_name = 'Employee'
with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()
    cursor.execute('pragma table_info({})'.format(table_name))
    names = [tup[1] for tup in cursor.fetchall()]
    print(names)
```


输出：

```
['id', 'name', 'date', 'salary', 'department']
```

即显示表的列名称。

8．行对象

默认情况下，数据库中的提取方法返回的值是表示数据库的“行”的元组。对于每一行，调用时都需要根据查询语句中列的顺序从元组中提取该行的各个值。

当查询中的值的数量增加或处理数据的代码在库中分散时，使用对象封装每一行的元组内容并以列名访问值通常更方便。当因修改查询语句导致返回的元组内容的数量和顺序发生变化时，程序中依赖查询结果的代码不会被破坏。

sqlite3 包含一个被用作 rowfactory 的 Row 类，将 sqlite3 数据库的连接(Connection)对象具有 rowfactory 属性设置为 sqlite3.Row：

```
conn.row_factory = sqlite3.Row
```

可以将查询结果集中的每一行转变为一个对象类型，然后可以使用列索引或名称查询每一行中的某个列的值。

```
import sqlite3
db filename = 'company.db'
with sqlite3.connect(db_filename) as conn:
    #修改 row factory 以便使用 Row
    conn.row_factory = sqlite3.Row  #设置该标志，使查询结果的每一行都可以通过列名查询
    #执行 SQL 查询语句
    cursor = conn.cursor()
    cursor.execute("""
     select name, date, description from Department
     where name = 'Sell'
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        """)
        name, date, description = cursor.fetchone() #得到查询结果的一行
        print('部门细节 for {} \n ({}) {}'.format(
            name, date, description))

        #执行 SQL 查询语句
        cursor.execute("""
        select id, name, date, salary from Employee
        where department = 'Sell' order by date
        """)

        print('\n 5 个雇员:')
        for row in cursor.fetchmany(5):
            print('{:2d} [{:<9}] ({}) [{:<8}] '.format(
                row['id'], row['name'], row['date'], row['salary'], #通过列名访问每行的列值
            ))
```

最后，循环中的函数 print() 中通过列名可以得到每一行中对应该列的值，如 row['id']得到行 row 的对应列名“id”的值。

输出：

```
部门细节 for Sell
 (2008-01-05) sell productions

 5 个雇员:
 2 [Zhang ping] (2008-01-05) [7000.5 ]
```

现在，打印语句会使用“列”关键字查找每一行中的列值，而无论查询中的列的排列顺序是什么。

9. 添加列

可以通过 SQL 语句 alter 给一个表添加一列，默认每一行的这个列的值为空值(NULL)，也可设置一个默认值。例如：

```
import sqlite3
db_filename = 'company.db'
table_name='Employee'
new_column1 = 'phone'
new_column2 = 'mobile'
column_type = ' TEXT'
default_val = '15370200000'
with sqlite3.connect(db_filename) as conn:
    #修改 row factory 以便使用 Row
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()
    #A) 增加一个没有默认值的新列
    cursor.execute("ALTER TABLE {tn} ADD COLUMN '{cn}' {ct}"\
        .format(tn=table_name, cn=new_column1, ct=column_type))

   #B) 增加一个有默认值的新列
    cursor.execute("ALTER TABLE {tn} ADD COLUMN '{cn}' {ct} DEFAULT '{df}'"\
        .format(tn=table_name, cn=new_column2, ct=column_type, df=default_val))
    conn.commit()

    cursor.execute('pragma table_info({})'.format(table_name))
    #将每一行的 tuple 放入一个 list 对象
    names = [tup[1] for tup in cursor.fetchall()]
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    print(names)

    cursor.execute("""
    select id, name, date, salary,phone,mobile from Employee
    where department = 'Sell'
    """)

    for row in cursor.fetchall():
        Employee_id, name, date, salary,phone,mobile = row
        print('{:2d} {:<25} ({}) [{:<8}] ({}) ({})'.format(
            Employee_id, name, date, salary,phone,mobile) )
```

分别给表“Employee”增加了“phone”和“mobile”这两个新列，“mobile”列有默认值15370200000。

程序输出：

```
['id', 'name', 'date', 'salary', 'department', 'phone', 'mobile']
 2 Zhang ping              (2008-01-05) [7000.5  ] (None) (15370200000)
```

15.5.2 在查询中使用变量

如果程序中 SQL 的各种 Query 语句全部用字面量定义则操作不灵活，针对一个数据库表的 Query 语句不能用于查询另外的数据库表的问题，SQL 语句支持“占位符”，通过将主机变量 host variables 动态地传递给占位符，可以使 SQL 语句根据这些变量的值执行不同的 SQL 操作。

SQL 有两种占位符，“位置”参数和“命名”参数。

1. “位置”参数

SQL 语句中的?表示“位置”占位符，在调用游标的函数 execute()时，可以通过一个 tuple 对象，将所有位置占位符传递给 SQL 子句。例如：

```
import sqlite3
import sys
db_filename = 'company.db'
department_name = 'Sell' #sys.argv[1]      #变量 department_name

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    query = """
    select id, name, date, salary,phone,mobile from Employee
    where department = ?
    """
    cursor.execute(query, (department_name,))   #通过一个 tuple 对象将变量传给占位符

    for row in cursor.fetchall():
        Employee_id, name, date, salary,phone,mobile = row
        print('{:2d} {:<25} ({}) [{:<8}] ({}) ({})'.format(
            Employee_id, name, date, salary,phone,mobile) )
```

输出：

```
2 Zhang ping              (2008-01-05) [7000.5  ] (None) (15370200000)
```

如果要将同一个 SQL 指令用于大量的数据，则可以调用游标的函数 executemany()一次处理多条记录，以避免在 Python 代码中使用一个循环不断地执行 SQL 指令，从而提高程序的效率。例如：

```
import sqlite3
import sys
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
db_filename = 'company.db'
department_name = 'Sell' #sys.argv[1]          #变量

Employees = (
    ('Li Ping','2010-02-05', 5000.8,'Research'),
    ('Zhang Wei', '2000-02-05', 1005.8,'Sell'),
    ('Wang Chong', '1989-02-05', 005.8,'Sell'),
    ('Zhao Si', '2015-02-05', 3005.8,'Sell'),
    ('Liu Neng', '2010-02-05', 4005.8,'Research'),
    ('Qiao Shi', '2018-02-05', 5005.8,'Sell'),
    ('Sang Tian', '2001-02-05', 8005.8,'Research')
)

with sqlite3.connect(db_filename) as conn:
    cur = conn.cursor()

    cur.execute("DROP TABLE IF EXISTS Employee")
    cur.execute("""
           CREATE TABLE Employee
            (id  integer primary key autoincrement not null,
            name   text, date text, salary real,
            department text not null references Department(name))
           """)
    #将 Employees 中的值传递给占位符，一次性插入多条记录
    cur.executemany("INSERT INTO Employee (name, date, salary, department)
    VALUES(?, ?, ?,?)", Employees )

    conn.commit()

    query = """
    select id, name, date, salary from Employee
    where department = ?
    """
    cursor.execute(query, (department_name,))   #通过一个 tuple 对象将变量传递给占位符
    for row in cursor.fetchall():
        Employee_id, name, date, salary = row
        print('{:2d} {:<25} ({}) [{:<8}] '.format(
            Employee_id, name, date, salary) )
```

输出：

```
 2 Zhang Wei               (2000-02-05) [1005.8  ]
 3 Wang Chong              (1989-02-05) [5.8     ]
 4 Zhao Si                 (2015-02-05) [3005.8  ]
 6 Qiao Shi                (2018-02-05) [5005.8  ]
```

2. 命名参数

对于具有大量参数的更复杂查询或在查询中多次重复某些参数的情况，可以使用命名参数。命名参数以冒号:开头(如：param_name)。例如：

```
import sqlite3
import sys

db_filename = 'company.db'
department_name = 'Sell' #sys.argv[1]        #变量

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        #:Department_name 定义了一个命名参数 Department_name
        query = """
        select id, name, date, salary from Employee
        where department = :Department_name
        order by name, date
        """
       #命名参数：Department_name 的值是变量 department_name
        cursor.execute(query, {'Department_name': department_name})
        for row in cursor.fetchall():
            employee_id, name, date, salary = row
            print('{:2d} [{:<10}]  ({})  [{:<8}] '.format(
                employee_id, name, date, salary))
```

输出：

```
 6 [Qiao Shi  ]  (2018-02-05)  [5005.8  ]
 3 [Wang Chong]  (1989-02-05)  [5.8    ]
 2 [Zhang Wei ]  (2000-02-05)  [1005.8  ]
 4 [Zhao Si   ]  (2015-02-05)  [3005.8  ]
```


SQL 的 Query 参数也可以用于 select、insert 和 update 语句，Query 参数可以用于任何字面量合法的地方。

```
import sqlite3
import sys

db_filename = 'company.db'
name = 'Qiao Shi'   #sys.argv[1]
salary = 4356.9   #float(sys.argv[2])

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()
    query = "update Employee set salary = :salary where name = :name"
    cursor.execute(query, {'name': name, 'salary': salary})

    conn.commit()

    query = """
    select id, name, date, salary from Employee
    where department = ?
    """
    cursor.execute(query, (department_name,)) #通过一个 tuple 对象将变量传递给占位符

    for row in cursor.fetchall():
        Employee_id, name, date, salary = row
        print('{:2d} {:<25} ({}) [{:<8}] '.format(
            Employee_id, name, date, salary) )
```

输出：

```
2 Zhang Wei                 (2000-02-05) [3005.8  ]
3 Wang Chong                (1989-02-05) [2005.8  ]
4 Zhao Si                   (2015-02-05) [5005.8  ]
6 Qiao Shi                  (2018-02-05) [4356.9  ]
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

15.5.3 事务

1. 事务

可能有来自不同客户或同一个程序的多个连接共同访问一个数据库，通过一个连接对数据库可进行多项更改，那么此时如何保证这些操作互不干扰、不会破坏数据库的完整性呢？

数据库管理系统的一个关键特性就是使用事务维护一致的内部状态，而一个事务中可包含多项对数据的操作(包括更改)。事务拥有以下四个特性。

- 原子性：数据库启动事务功能后，每个事务的所有 SQL 操作都能作为一个整体以原子的方式执行，即其中的 SQL 操作要么全部执行，要么一个也不执行。
- 一致性(Consistency)：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态是指数据库中的数据应满足完整性约束。
- 隔离性(Isolation)：多个事务并发执行时，一个事务的执行并不应影响其他事务的执行。
- 持久性(Durability)：已被提交的事务对数据库的修改应该永久保存在数据库中。

当事务被提交给了 DBMS(数据库管理系统)，则 DBMS 需要确保该事务中的所有操作都被成功完成且其结果被永久保存在数据库中，如果事务中某些操作没有成功完成，则事务中的所有操作都需要被回滚，并回滚到事务执行前的状态。一个事务对数据库其他操作或其他事务的执行并无影响，所有的事务都好像在独立地运行。

sqlite3 的事务处理很简单，就是通过 commit()方法提交事务，通过 rollback()方法回滚事务(放弃事务里的所有操作)。

2. 用 commit()方法提交事务

只有显式调用或隐式调用 commit()方法，此前的 SQL 操作在数据库中才真正生效。例如，Python 的 with 语句会隐式地自动调用连接的 close()，而 close()会自动调用连接(connection)对象的 commit()方法。

下面的代码中模拟三个数据库连接，以说明事务处理的提交(commit()方法)功能的作用，第一个连接执行了一些 SQL 数据库操作，但并没有提交(commit()方法)，尽管在这个连接中通过游标对象可以查询修改过的数据记录，但第二个连接并不能看到数据库的变化。当提交后，第三个连接就能看到这些修改。

```
import sqlite3
db_filename = 'company.db'
def show_departments(conn):
    cursor = conn.cursor()
    cursor.execute('select name,date, description from Department')
    for name,date, description in cursor.fetchall():
        print('  ', name,date)


with sqlite3.connect(db_filename) as conn1:
    print('修改前:')
    show_departments(conn1)

    #在 cusor()中插入
    cursor1 = conn1.cursor()
    cursor1.execute("""
    insert into Department (name, date, description)
    values ('Human',  '2017-01-01','人事部门')
    """)
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
    print('\n conn1 执行修改后:')
    show_departments(conn1)

    #从另外的连接(connection)查询, conn1 还没有提交操作
    print('\nconn1 提交前:')
    with sqlite3.connect(db_filename) as conn2:
        show_departments(conn2)

    #conn1 提交后, 从另外的连接(connection)查询
    conn1.commit()
    print('\nconn1 提交后:')
    with sqlite3.connect(db_filename) as conn3:
        show_departments(conn3)
```

程序输出:

```
修改前:
   Research 2006-01-05
   Sell 2008-01-05

 conn1 执行修改后:
   Research 2006-01-05
   Sell 2008-01-05
   Human 2017-01-01

conn1 提交前:
   Research 2006-01-05
   Sell 2008-01-05

conn1 提交后:
   Research 2006-01-05
   Sell 2008-01-05
   Human 2017-01-01
```


3. 放弃修改

可以用 rollback()方法放弃未提交的 SQL 操作。commit()方法和 rollback()方法通常分别被放置在 try、except 子句的不同部分，当出现错误时，可以调用 rollback()方法放弃未提交的修改。例如:

```
import sqlite3
db_filename = 'company.db'
def show_departments(conn):
    cursor = conn.cursor()
    cursor.execute('select name,date, description from Department')
    for name,date, description in cursor.fetchall():
        print('  ', name,date)

with sqlite3.connect(db_filename) as conn:
    print('修改前:')
    show_departments(conn)
    try:
        #将数据插入 cursor()
        cursor = conn.cursor()
```

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
            cursor.execute("""delete from Department
                        where name = 'Human'
                        """)
            #显示插入的结果
            print('\n 删除后:')
            show_departments(conn)

            #模拟处理过程中出错
            raise RuntimeError('simulated error')
        except Exception as err:
            #出现异常，舍弃修改
            print('ERROR:', err)
            conn.rollback()   #回滚
        else:
            #没有出现异常情况下，提交修改
            conn.commit()

        #显示结果
        print('\nrollback 回滚后:')
        show_departments(conn)
```


程序输出：

```
修改前:
   Research 2006-01-05
   Sell 2008-01-05
   Human 2017-01-01

删除后:
   Research 2006-01-05
   Sell 2008-01-05
ERROR: simulated error

rollback 回滚后:
   Research 2006-01-05
   Sell 2008-01-05
   Human 2017-01-01
```

4．隔离级别

sqlite3 支持三种锁定模式，称为**隔离级别**(Isolation Levels)。隔离级别主要用于防止不同连接之间不兼容事务操作的不同控制方式。

如下代码所示，打开一个数据库连接时通过向形参 isolation_level 传递代表隔离级别的实参来设置隔离级别。

```
sqlite3.connect(filename, isolation_level=isolation_level)
```

有四个不同的隔离级别值，Deferred、Immediate、Exclusive、Autocommit。如图 15-1 所示是 SQL 语句的语法图，可以看出不同隔离级别对事务的影响，isolation_level 参数决定开启事务时使用的是 BEGIN TRANSACTION, BEGIN DEFERRED TRANSACTION, BEGIN IMMEDIATE TRANSACTION, BEGIN EXCLUSIVE TRANSACTION 中的哪一种隔离级别。

isolation_level 默认为 None，表示自动提交(Autocommit)。不同的数据库连接可以使用不同的隔离级别值。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

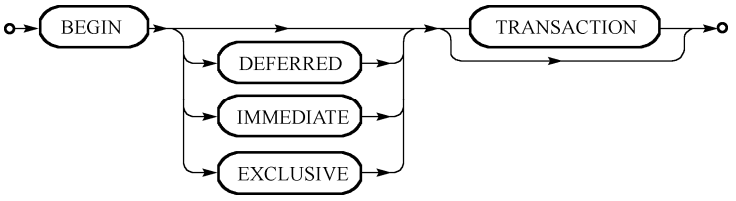


图 15-1　SQL 语句的语法图

下面的程序用于演示不同隔离级别对使用单独连接到相同数据库的线程中事务顺序的影响，程序中共有四个线程，其中两个线程通过更新现有的行并将其更改写入数据库，其他两个线程尝试读取 Employee 表中的所有行。

```
import logging, sqlite3, sys, threading, time
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s (%(threadName)-10s) %(message)s',
)

db_filename = 'company.db'
isolation_level = None  # 'DEFERRED' # sys.argv[1]
def writer():
    with sqlite3.connect(
           db_filename,
           isolation_level=isolation_level) as conn:
        cursor = conn.cursor()
        cursor.execute('update Employee set salary = salary + 1000')
        logging.debug('等待同步')
        ready.wait()      # 同步线程
        logging.debug('暂停')
        time.sleep(1)
        conn.commit()
        logging.debug('提交了修改')

def reader():
    with sqlite3.connect(
           db_filename,
           isolation_level=isolation_level) as conn:
        cursor = conn.cursor()
        logging.debug('等待同步')
        ready.wait()   # 同步线程
        logging.debug('等待结束')
        cursor.execute('select * from Employee')
        logging.debug('执行了 SELECT 查询')
        cursor.fetchall()
        logging.debug('得到了查询结果')

if __name__ == '__main__':
    ready = threading.Event()

    threads = [
        threading.Thread(name='Reader 1', target=reader),
        threading.Thread(name='Reader 2', target=reader),
        threading.Thread(name='Writer 1', target=writer),
        threading.Thread(name='Writer 2', target=writer),
    ]
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
        [t.start() for t in threads]

        time.sleep(1)
        logging.debug('准备设置锁')
        ready.set()

        [t.join() for t in threads]
```

程序使用线程模块 threading 中的 Event 对象(ready = threading.Event())同步线程。函数 Writer()连接并更改数据库，等待同步事件发生(ready.wait())，然后休眠 1 秒(time.sleep(1))，再将更新提交数据库。函数 reader()连接数据库，然后等待直到同步事件发生后查询数据库。

使用“延迟锁”(Deferred)：这种模式是 sqlite3 的默认模式，即只在发生改变时才会锁上被更新的记录。

执行命令行：

```
>>>python isolation.py DEFERRED
```

输出结果：

```
2018-10-06 14:33:01,698 (Reader 1  ) 等待同步
2018-10-06 14:33:01,700 (Reader 2  ) 等待同步
2018-10-06 14:33:01,705 (Writer 1  ) 等待同步
2018-10-06 14:33:02,706 (MainThread) 准备设置锁
2018-10-06 14:33:02,714 (Reader 1  ) 等待结束
2018-10-06 14:33:02,715 (Reader 1  ) 执行了 SELECT 查询
2018-10-06 14:33:02,714 (Writer 1  ) 暂停
2018-10-06 14:33:02,714 (Reader 2  ) 等待结束
2018-10-06 14:33:02,720 (Reader 1  ) 得到了查询结果
2018-10-06 14:33:02,722 (Reader 2  ) 执行了 SELECT 查询
2018-10-06 14:33:02,727 (Reader 2  ) 得到了查询结果
2018-10-06 14:33:03,902 (Writer 1  ) 提交了修改
2018-10-06 14:33:03,966 (Writer 2  ) 等待同步
2018-10-06 14:33:03,969 (Writer 2  ) 暂停
2018-10-06 14:33:05,145 (Writer 2  ) 提交了修改
```

使用“立即锁”(Immediate)：在这种模式下，只要更新数据库，就会立即锁上这条记录，直到事务提交才会打开锁。

执行命令行：

```
>>>python isolation.py IMMEDIATE
```

输出结果：

```
2018-10-06 14:32:10,977 (Reader 1  ) 等待同步
2018-10-06 14:32:10,979 (Reader 2  ) 等待同步
2018-10-06 14:32:10,983 (Writer 1  ) 等待同步
2018-10-06 14:32:11,982 (MainThread) 准备设置锁
2018-10-06 14:32:11,982 (Reader 1  ) 等待结束
2018-10-06 14:32:11,983 (Reader 2  ) 等待结束
2018-10-06 14:32:11,983 (Writer 1  ) 暂停
2018-10-06 14:32:11,986 (Reader 1  ) 执行了 SELECT 查询
2018-10-06 14:32:11,989 (Reader 2  ) 执行了 SELECT 查询
2018-10-06 14:32:11,995 (Reader 1  ) 得到了查询结果
2018-10-06 14:32:11,997 (Reader 2  ) 得到了查询结果
2018-10-06 14:32:13,167 (Writer 1  ) 提交了修改
```


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

```
2018-10-06 14:32:13,243 (Writer 2  ) 等待同步
2018-10-06 14:32:13,244 (Writer 2  ) 暂停
2018-10-06 14:32:14,421 (Writer 2  ) 提交了修改
```

使用“排他锁”(Exclusive)：这种锁会对所有的读/写操作都上锁。一般用于对数据库性能要求较高的情况，因为一旦上锁，这个数据库连接就只能为一个使用者使用。

执行命令行：

```
>>>python isolation.py EXCLUSIVE
```

输出结果：

```
2018-10-06 14:31:24,953 (Reader 1  ) 等待同步
2018-10-06 14:31:24,954 (Reader 2  ) 等待同步
2018-10-06 14:31:24,962 (Writer 2  ) 等待同步
2018-10-06 14:31:25,949 (MainThread) 准备设置锁
2018-10-06 14:31:25,949 (Reader 1  ) 等待结束
2018-10-06 14:31:25,949 (Reader 2  ) 等待结束
2018-10-06 14:31:25,949 (Writer 2  ) 暂停
2018-10-06 14:31:27,100 (Writer 2  ) 提交了修改
2018-10-06 14:31:27,103 (Writer 1  ) 等待同步
2018-10-06 14:31:27,105 (Writer 1  ) 暂停
2018-10-06 14:31:28,264 (Writer 1  ) 提交了修改
2018-10-06 14:31:28,318 (Reader 2  ) 执行了 SELECT 查询
2018-10-06 14:31:28,318 (Reader 2  ) 得到了查询结果
2018-10-06 14:31:28,319 (Reader 1  ) 执行了 SELECT 查询
2018-10-06 14:31:28,319 (Reader 1  ) 得到了查询结果
```


使用“自动锁”(Autocommit)：可以把锁级别设置为 None，即自动提交模式。每次对数据库的修改都会自动提交到数据库。因为是自动提交，所以应在代码中删除“conn.commit()”，并将代码中 isolation_level 的值设置为 None。

执行命令行：

```
>>>python isolation.py
```

输出结果：

```
2018-10-06 14:29:45,740 (Reader 1  ) 等待同步
2018-10-06 14:29:45,749 (Reader 2  ) 等待同步
2018-10-06 14:29:45,922 (Writer 1  ) 等待同步
2018-10-06 14:29:46,100 (Writer 2  ) 等待同步
2018-10-06 14:29:46,759 (MainThread) 准备设置锁
2018-10-06 14:29:46,770 (Reader 1  ) 等待结束
2018-10-06 14:29:46,773 (Reader 2  ) 等待结束
2018-10-06 14:29:46,773 (Writer 1  ) 暂停
2018-10-06 14:29:46,773 (Writer 2  ) 暂停
2018-10-06 14:29:46,779 (Reader 1  ) 执行了 SELECT 查询
2018-10-06 14:29:46,786 (Reader 1  ) 得到了查询结果
2018-10-06 14:29:46,786 (Reader 2  ) 执行了 SELECT 查询
2018-10-06 14:29:46,790 (Reader 2  ) 得到了查询结果
2018-10-06 14:29:47,783 (Writer 1  ) 提交了修改
2018-10-06 14:29:47,786 (Writer 2  ) 提交了修改
```

总结

- sqlite3 模块是底层的 C 语言数据库 SQLite 的 Ptyhon 接口。
- 通过一个数据库的连接的 Cursor 对象(游标对象)可对数据库执行各种 SQL 操作（创建、插入、查询、更新、删除等)。
- 可使用位置或命令参数更灵活地操作数据库。
- commit()和 rollback()分别用于提交事务、回滚事务，并可设置事务处理的隔离级别。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

参 考 文 献

1. Python 官方文档. https://docs.python.org/3/
2. https://docs.python.org/3/tutorial/index.html
3. https://www.programiz.com/python-programming/tutorial
4. Google's Python Class. https://developers.google.com/edu/python/
5. https://python-textbok.readthedocs.io/
6. TkInter 库. https://wiki.python.org/moin/TkInter
7. Doug Hellmann.Python 3 Module of the Week. https://pymotw.com/3/
8. https://realpython.com/python-sockets/
9. Stavros Korokithakis. Tutorial - Learn Python in 10 minutes. https://www.stavros.io/tutorials/python/
10. https://hwdong-net.github.io

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888
传　　真：（010）88254397
E-mail：　dbqq@phei.com.cn
通信地址：北京市海淀区万寿路 173 信箱
　　电子工业出版社总编办公室
邮　　编：100036

電子工業出版
PUBLISHING HOUSE OF ELECTRONICS I

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY